Desenvolvimento Baseado em Modelos 2016/2017

Projeto Fase 1 - Relatório Técnico

Turma 3ºINF ES-03

Profº João Ventura

Miguel Furtado – 120221006

Conteúdo

1. Desenvolvimento em laboratório

1.1 Descrição do Projeto

O projeto implementado tem como intuito a geração de código automático a partir de modelos. Nesta fase pretende-se a implementação de diversas classes de java automaticamente, a implementação de uma base de dados funcional relativa às classes criadas e as relações entre estas.

Para este fim foi necessário a introdução de uma API que se chama freemarker. Esta API tem como propósito gerar texto em output, mas este texto pode ser alterado para ir de encontro ao que necessitamos. Para isto são criados os templates. Permitem-nos assim gerar texto para criar qualquer tipo de documento desde que o ficheiro ".ftl" esteja estruturado para esse tipo de ficheiro.

Definiu-se algumas classes que são os nossos meta-modelos. Estes irão regular o comportamento dos objetos usados para a criação de ficheiros ".java", que serão gerados automaticamente, e fornecer um maior nível de abstração. Inicialmente definimos as classes Model, Class e Attribute. Estas são a base do desenvolvimento do projeto e a partir destas é que vamos gerar código, ou seja, a partir desta vamos permitir ao utilizador criar um conjunto de elemento do tipo Class (Model), e definir o nome do objeto Class e que Attributes tem. Os objetos Model criados serão enviados para os respetivos templates que irá gerar os ficheiros ".java".

Foi então introduzido um novo template. Este template permite a geração de base de dados com base nas classes do tipo Class que cada Model contem. Foi assim possível guardar informação numa base de dados. Esta permite a criação de ligações entre objetos com relações. Sendo possível criar relações 1 para 1, 1 para n e n para n.

De seguida foi introduzido a capacidade de validar um ficheiro ".xml". Para isto foi criado um novo meta-modelo que iria validar um ficheiro ".xml". Este se considerado válido poderia então ser transformado para um ficheiro ".java" com respetiva base de dados associada. Este tipo de processo intitula-se de Transformação Modelo para Modelo.

Este tipo de processo é uma opção que o utilizador pode escolher, para que a geração das classes para o seu programa e respetivas bases de dados com relações, seja criado automaticamente.

1.2 Package metamodels

O Package metamodels vai ser o pacote que guarda todos os nossos metamodelos. Os metamodelos vão definir como será o comportamento do que for definido nesses metamodelos. Dado vamos precisar de criar modelos foi então criado o Model.java que vai definir como é o Model. O mesmo se aplica para a Class.java e Attribute.java. Temos também um model.dtd que irá definir qual o aspeto do xml que contenha uma ligação ao dtd.

1.2.1 Model.java

A classe Model vai conter um nome, uma lista do tipo Class de classes e uma lista do tipo Relation, para definir as relações que existem entre as classes ao n

O Modelo é o core da aplicação pois é a partir deste que vamos gerar o código. O conteúdo desta passará por vários processos para a geração do código automaticamente. Sendo que a lista de classes que este contem dará origem a ficheiros ".java". As relações definidas no modelo serão usadas para que a geração das tabelas em base de dados seja a mais correta possível e possibilite guardar e aceder sem qualquer problema.

1.2.2 Class.java

Esta class irá servir para criar os objetos java. Este irão definir o comportamento dos objetos criados.

Este objeto vai ter como atributos, um nome, uma lista de atributos do Tipo Attibute, uma lista de atributes required do tipo Attribute, uma lista de string chamada foreignKeys, uma List do tipo Relation que se chama relations, uma String pkg e um parente.

Todos estes serão manipulados para que no fim o ficheiro java gerado esteja o mais correto possível.

Alguns métodos importantes:

- **public void addAttribute(Attribute attribute)** – Este terá como objetivo adicionar um attributo à lista de atributos, mas caso o atributo seja required, então será adicionado, também, à lista de required.

- **public void addForeignKey(String name) –** Será para adicionar uma foreignKey à lista de foreignKeys

- **public String getForeignKey(int id) –** Serve para ir buscar uma foreignKey a partir de um id.

1.2.3 Attribute.java

A class Attribute terá como objetivo definir o que é um atributo de uma class. Um atributo pode ter um nome, um tipo e se este é required ou não. Este último vai definir se irá ser criado, na geração de código, um construtor esta para estes.

Este objeto será usado em todas os objetos do tipo Class e para definir os atributos desta.

1.2.4 model.dtd

O model.dtd tem como objetivo definir a estrutura de um ficheiro XML e consequentemente a sua validação. Assim podemos criar ficheiros XML automaticamente e o dtd irá validar se está de acordo, para além disto podem ser cirados ficheiros XML sem ser automaticamente e usados para a criação de classes java e respetivas base de dados.

Exemplo de um ficheiro XML válido:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE model SYSTEM "../metamodels/model.dtd">
<model name="Person">
    <class name="Person">
        <attribute name="name" type="String" required="true" />
        <attribute name="age" type="int" required="true"/>
        <relation name="car" type="1ToN" />
    </class>
</model>
```

1.3 Package models

Os ficheiros xml no package models têm como objetivo serem usados em transformações. Estes vão ser transformados num model com as respetivas classes descritas no xml, um ficheiro sql para os acessos à base de dados e os respetivos ficheiros java.

1.4 Package templates

Os templates são a base para a criação seja das classes das tabelas sql. O programa ao correr e ao fazer transformações vai passar nos templates para obter a string com a informação necessária para criar os ficheiros java e também o acesso à base de dados.

1.4.1 java_class.ftl

Este template vai servir para gerar as classes automaticamente com base no que a classe em si é composta. Automaticamente será gerada com alguns acessos à base de dados, save, delete, where, get.

1.4.2 java_model.ftl

Este template é para ser usado caso se queria gerar o modelo e as classes associadas a esse modelo. Este está ligado ao template java_class.ftl e por esta razão a partir deste é que é possível gerar as classes.

1.4.3 sqlite3_create.ftl

Este template é usado para criar tabelas para armazenar informação (DB). Este vai ser gerado com base no modelo e nas classes. Irá criar as tabelas para as respetivas classes com os seus respetivos atributos.

1.5 Package utils

Este package tem como objetivo guardar algumas coisas que nos possam ser úteis e que irão ser usadas em vários sítios do programa.

1.5.1 Package sqlite

1.5.1.1 SQLiteConn.java

Esta classe tem como objetivo fazer as ligações à base de dados. Executar querys que podem devolver um ResultSet com informação necessária e um execute update para serem efetuados inserts, updates e deletes.

1.5.2 Package transformations

1.5.2.1 M2M.java

Esta classe tem como objetivo fazer as transformações de modelo para modelo. Para tal pegamos num modelo, seja este um ficheiro xml ou xmi, e a partir desse criamos um novo modelo com uma base de dados.

1.5.2.2 Model2Text.java

Esta classe tem como objetivo pegar num modelo e gerar o texto necessário para criar os ficheiros java e de base de dados a partir da API freemarker. Com os templates definidos no Package template e a informação do model conseguimos gerar o mencionado com tudo o que necessitamos.

1.6 Package xmiModels

Este Package tem com objetivo guardar os ficheiros xmi, para que posteriormente possa ser acedido para a criação do modelo com uma transformação M2M com a ajuda da classe M2M.java no package transformations.

2. Extras

2.1 Package Menu

Inicialmente este package foi criado quando foi iniciado o desenvolvimento da interface. Mais Tarde as scenes da interface foram adicionadas a um package chamado Scene e este componente ficou apenas neste pacote. A ideia é mais tarde se alguns componentes individuais forem criados devem ser adicionados neste pacote.

2.1.1 BarMenu.java

Esta classe é componente para a interface gráfica. Este vai ser usado como a barra superior e por agora apenas tem a função de disponibilizar a opção de andar para trás entre as cenas. Mais tarde poderiam ser adicionadas mais funcionalidades a este menu.

2.2 Package metamodels

2.2.1 Class.java

Para os extras foi necessário que esta classe recebesse alguns atributos novos para a manipulação de dados e geração do código. Como tal alguns dos atributos novos são:

- **private int classId** – Este atributo foi adicionado para conseguirmos identificar a class numa base de dados, sendo pelo id da class que ficaria na tabela classe, mas também para os atributos saberem a que classe pertencem.

- **private List<Relation> relations** – Este atributo foi adicionado para que na criação dos modelos se pudesse especificar as relações que cada uma possui. Será útil para depois sabermos que foreign keys associar mais tarde quando fizermos as relações 1To1 e 1ToN.

- **Private List<Class> hiddenKeys** – Este atributo serve de apoio à foreign key, ou seja, quando uma classe tem uma foreign para outra existe uma relação. Se quisermos saber na classe que não tem foreign key a quem está associada precisamos de um indicar. Para esse fim foi criado este atributo que mesmo na classe, sem foreign key, sabe qual é a classe que tem uma associação. Assim podemos gerar os ficheiros ".java" com os métodos para aceder a informação na base de dados.

Com alguns métodos definidos precisamos também de um construtor vazio, pois na base de dados ao irmos buscar os dados da classe não vamos ter toda a informação necessária pois alguns dados já estão disponíveis pela utilização da interface.

Alguns métodos novos adicionados foram:

- **public void resolveRelation(String name, String type) -** Este método tem como objetivo remover relação a partir do nome da relação e do tipo de relação. A utilização deste método será para que as transformações tenham as relações corretas entre si. E se será usado na classe builder.java.

- **public String getForeignKey(int id) –** Método para obter uma foreign key a partir de um id.

- **public void addRelations(String name, String type) –** Serve para adicionar uma relação à lista de relações a partir do nome da relação e do tipo. Este método vai ser usado para definir relações na construção do model.

2.2.2 Model.java

Nesta classe para os extras foi adicionado um atributo:

- **private HashMap<Class, Class> relation –** Este atributo tem como objeitvo guarder apenas as relações N para N. Mais tarde será usado na geração de código para permitir a criação de uma tabela sql, algo que ficou por implementar e que também era o intuito, era acesso na classe gerada à base de dados.

Como adiciona-mos um atributo novo foi necessário acrescentar um método:

- **public void addRelation(Class clazz, Class clazz1) –** Este irá adicionar uma relação ao HashMap a partir da classe Builder.java.

2.2.3 Relation.java

A class Relations tem como objetivo definir o que é uma relação. Como tal terá como atributos:

- **private String name -** o nome da classe com quem se está a formar uma relação
- **private String type -** o tipo de relação, sendo esta última, 1To1, 1ToN ou NToN.

Este objeto em conjunto com a Lista de Relations usada na class Class e irá servir de apoio para lidar com as relações entre classes.

2.3 Package MyModels

O Package MyModels foi criado por existir a necessidade de se guardar os Modelos gerados apenas num sítio. Como os modelos gerados serão guardados nesta pasta foi então criada uma class para guardar os models e uma class para manipular os dados. Também.

2.3.1 DataManager.java

A class DataManager tem como intuito manipular a informação dos modelos na base de dados. Este servirá de apoio à base de dados. Como tal a maioria dos métodos vão receber dados relacionados com a interface.

Para a manipulação dos dados foram então criados vários métodos para esse fim que são os seguintes:

- **public ArrayList<String> all(String modelName, String className) throws SQLException -** Este método permite obter todos os atributos de uma classe. Para isso chama um método que lhe vai fornecer a informação que é o método **getResultSet().** Este método também se encontra nesta class.

- **public String getElementById(String modelName, String className, TextField textField) throws SQLException –** Método que permite ir buscar um element a partir do id. Este método vai ser usado para pesquisas sendo que o utilizador só terá que fornecer um id no textField.

- **public List<String> getElementsByICondition(String modelName, String className, TextField textField) throws SQLException –** Método para ir buscar elementos a partir de uma condição. O utilizador terá apenas que fornecer a condição na interface, com o modelo e a classe escolhida e poderá aceder aos dados a partir da base dde dados.

- **public void save(String modelName, String className, ArrayList<Attribute> objectAttributes, List<String> listaValores) –** Método para gravar um atributo na base de dados. Assim mais tarde será possível aceder aos objetos criados para cada classe pois estarão guardados na base de dados.

- **public void deleteModel(int index, String modelName) –** Método para apagar um modelo. Quando apagamos um modelo é necessário apagar da base de dados. Para de apagarmos o modelo apagamos também as classes associadas.

- **public void deleteClass(int modelIndex, int classIndexDB, int classIndexLocal, String modelName, String className, List<Model> myModels) –** Método para remover uma classe. Ao remover uma classe esta será apagada da base de dados e também do modelo que está na nossa lista de modelos.

- **public void deleteAttribute(String modelName, String className, int attributeId) –** Método para apagar um objeto de uma classe. Este será apagado na base de dados.

- **public static void deleteDirectory(File element) –** Método para pagar um package. Quando queremos apagar um modelo este não pode existir mesmo, como tal também é apagado o package referente ao mesmo juntamente com todos os ficheiros que este contem.

- **public static void deleteFile(File element) –** Método para apagar um ficheiro. No caso de ser necessário apagar uma classe também será preciso apagar o ficheiro local. Como tal este método trata desses casos.

- **public void saveModelWithClass(Model model) –** Método para gravar um model com as respetivas classes na base de dados.

- **public void saveModel(Model model) –** Método para gravar um modelo na base de dados.

- **public void saveClass(Model model) –** Método para gravar uma classe na base de dados.

- **public ArrayList<Attribute> getColumnNumber(String modelName, String className) throws SQLException –** Método tem como objetivo devolver uma lista de atributos da classe.

- **public List<Model> getModelResultSet(String filename, String query) throws SQLException –** Método genérico que tem como objetivo ir buscar um model a partir de um caminho e de uma query. No fim vai devolver a lista de modelos existentes na base de dados.

- **public List<Class> getClassResultSet(String filename, String query) throws SQLException –** Método usado para ir buscar as classes relativas a um modelo à base de dados. Para além de ir buscar buscar as classes este também vai buscar os atributos à base de dados.

- **private ArrayList<String> getResultSet(String filename, String query) throws SQLException –** Método genérico que a partir de um caminho e de uma query vai buscar uma string com a informação que necessitamos.

2.3.2 MyModels.java

A class MyModels é onde vamos guardar a nossa lista de models localmente. Para isso foram necessários criar dois atributos:

- **private List<Model> myModels –** Este atributo é a nossa lista de modelos.
- **private DataManaget dataManager –** Este atributo é para facilitar o acesso à base de dados e manipulação.

Com os atributos foram necessários alguns métodos:

- **public void addModel(Model model) –** Este método permite adicionar um modelo à lista de modelos. Com o apoio do dataManager vai guardar a informação na base de dados.

- **public void removeModel(Model model)-** Método permite remover um modelo. Ao removermos para além de recorremos à nossa lista para remover vamos também usar o dataManager para apagar da base de dados.

- **public List<Model>allModels() throws SQLException –** Método que com o apoio do dataManager vai buscar uma lista de modelos.

- **public List<Class> allClasses() throws SQLException –** Método que com o apoio do dataManager vai buscar a lista de classes.

- **Public List<Model> getModels() throws SQLException –** Método que vai pegar na lista de modelos e classes obtidos nos dois métodos anteriores e criar uma Lista de modelos completo.

2.3.3 tables.db

Este ficheiro é a nossa base de dados e será de apoio à interface. A partir desta podemos aceder aos modelos criados classes a atributos. Com esta podemos aceder às tabelas específicas dos modelos criados.

2.4 Package sample

O Package sample foi o package criado automaticamente para se puder criar uma interface em javaFX logo a destacar a única coisa que foi usada neste package foi a classe Main.java.

2.4.1 Main.java

A classe Main tem como objetivo arrancar com o projeto. Esta classe para além de lançar a interface também disponibiliza alguns métodos para testar o ORM sem ser com a interface. Para isso temos 2 métodos:

- **public static void testORM1To1() throws SQLException –** Este método está feito para manipular objetos cujas relações são de

1To1. Será usado com a informação do modelo BookStore que já existe no projeto.

- **public static void testeORM1ToN() throws SQLException –** Este método está feito para manipular objetos cujas relações são de 1ToN. Será como o método anterior usado com a informação do modelo BookStore que já existe no projeto.

2.5 Package Scene

O Package Scene é o pacote que contem a informação sobre a interface, sendo esta as scenes todas e os componentes necessários para efetuar a criação e manipulação de modelos. Este package não será descrito em tanto detalhe.

2.5.1 ImportScene.java

A class ImportScene é a class associada à interface para se importar um ficheiro. Esta terá apenas um layout em que podemos escolher qual o tipo de ficheiro que queremos importar.

2.5.2 MyModelsScene.java

A class MyModelsScene é a class da interface, pois é nesta onde iremos realmente criar e manipular os nossos modelos e esta estará ligada a outras interfaces que apenas serão "filhas" desta.

Algo que se pode destacar nesta classe é o botão build model. Pois este é que vai gerar o nosso código automático a partir de um modelo. Para fazer build de um modelo o modelo necessita de ter classes apenas. Com o build feito, só então é que podemos usufruir da criação de objetos para as classes e também os acessos à base de dados. Pois antes não existem ficheiros para suportar essas ações.

2.5.3 NewCreateClassScene.java

A class NewCreateClassScene permite criar uma scene nova quando pretendemos adicionar uma classe ao modelo. Esta vai disponibilizar com base no metamodelo quais as coisas que o utilizador pode fazer para adicionar uma class.

2.5.4 NewModelScene.java

A class NewModelScene permite criar um novo modelo. Esta scene apenas vai disponibilizar algo para o utilizador inserir o input do nome do modelo e de seguida a sua criação.

2.5.5 PreMadeScene.java

A class PreMadeScene tem como objetivo disponibilizar a cena para a criação de modelos previamente construídos. O utilizador ao aceder a esta cena poderá escolher de entre 2 modelos que já existem, Person e BookStore e gerar os ficheiros automaticamente parra cada modelo.

2.5.6 SceneChanger.java

A class SceneChanger é a class que permite ao utilizador navegar entre os 3 menus. Conforme o que o utilizador necessite irá envia-lo para a scene respetiva.

2.6 Package templates

Para os extras tornou-se necessário acrescentar algumas coisas aos templates para manipulação de dados. Para além de acrescentar algumas coisas novas foi também criado o template xml_create.ftl. Este irá permitir que na geração do modelo seja também gerado automaticamente com os ficheiros java e a base de dados um xml com a informação sobre o modelo já corrigida.

2.6.1 java_class.ftl

Esta class com os extras necessitou de várias alterações. A primeira alteração foi nos atributos. Caso exista um foreign key criou-se a necessidade de se adicionar esta como atributo. Tendo isto em conta foi-se verificar onde haveria necessidade de foreing key e também caso existe é adicionada ao construtor que permite criar um objeto com todos os atributos. Se existem foreign keys podem precisar de sets e gets então também se adicionou.

Se existe um foreign Key então podemos através desta nova classe gerada ir buscar informação à nossa base de dados através da mesma e para isso foram criados alguns métodos que só são gerados caso essa exista. Se existe foreign key então também existe associado uma hidden key. Então surgiu o seguinte texto que poderá ser gerado para um ou dois métodos:

```
<#if hiddenKey?has_content>
<#list hiddenKey as hk>
<#if hk.foreignKeys?has_content>
/**
 *Method to get the elements in case exists an 1 to N or N to N relation
 */
public ArrayList<${hk.name}> getAll${hk.name}(int fk_${name?lower_case}Id)
throws SQLException{

   SQLiteConn sqLiteConn = new SQLiteConn("src/MyModels/${pkg}/tables.db");
   ResultSet rs = sqLiteConn.executeQuery("SELECT * FROM ${hk.name} WHERE
FK_${name?lower_case}Id = " + fk_${name?lower_case}Id);
   ArrayList<${hk.name}> ${hk.name?lower_case}List = new ArrayList<>();

   if (rs.getMetaData().getColumnCount() == 0) {
```

```
      System.out.println("No Results");
      return null;
    }

    while (rs.next()) {
        ${hk.name} ${hk.name?lower_case} = new
${hk.name}(rs.getInt(rs.getMetaData().getColumnName(1)),
        <#if hk.attributes?has_content>
        <#list hk.attributes as attribute>
        <#compress><#if attribute.type ==
"String">rs.getString(rs.getMetaData().getColumnName(${attribute?index + 2}))
        <#elseif attribute.type ==
"int">rs.getInt(rs.getMetaData().getColumnName(${attribute?index + 2}))
        <#elseif attribute.type ==
"Date">rs.getDate(rs.getMetaData().getColumnName(${attribute?index + 2}))
        <#elseif attribute.type ==
"double">rs.getDouble(rs.getMetaData().getColumnName(${attribute?index +
2}))</#if></#compress><#sep>,
         </#list>);
        </#if>
        <#if hk.foreignKeys?has_content>

${hk.name?lower_case}.get${name}().set${name}Id(fk_${name?lower_case}Id);
        </#if>

    ${hk.name?lower_case}List.add(${hk.name?lower_case});

    }

    sqLiteConn.close();
    return ${hk.name?lower_case}List;
}
</#if>
 /**
 * Method to save information regarding the fk ${hk.name}, this will save in the class
Book the fk of the ${name}
 */
 public void save${hk.name}() {

    SQLiteConn sqLiteConn = new SQLiteConn("src/MyModels/BookStore/tables.db");
    <#if foreignKeys?has_content>
     if(this.${hk.name?lower_case}.get${hk.name}Id() == -1){
            return;
        }
     </#if>
        sqLiteConn.executeUpdate("UPDATE ${name} SET " +
                    <#list attributes as attribute>
                    <#if foreignKeys?has_content>
```

```
                    <@compress single_line=true>" ${attribute.name?cap_first} =
                    <#if attribute.type == "String">'" + this.${attribute.name} + "'
                    <#else>" + this.${attribute.name} + "
                    </#if>, </@compress> " +
                    <#else>
                    <@compress single_line=true>" ${attribute.name?cap_first} =
                    <#if attribute.type == "String">'" + this.${attribute.name} + "'
                    <#else>" + this.${attribute.name} + "
                    </#if><#sep> , </@compress> " +
                    </#if>
                    </#list>
                    <#list foreignKeys as fk>
                    <@compress single_line=true>"Fk_${fk}Id = " +
this.${fk}.get${fk?cap_first}Id() +
                    <#sep> , </@compress>
                    </#list>
                    " WHERE ${name?lower_case}Id =" + ${name?lower_case}Id);

        <#list foreignKeys as fk>
         sqLiteConn.executeUpdate("UPDATE ${fk?cap_first} SET " +
                    <#list hk.attributes as attribute>
                    <@compress single_line=true>" ${attribute.name?cap_first} =
                    <#if attribute.type == "String">'" +
this.${fk?lower_case}.get${attribute.name?cap_first}() + "'
                    <#else>" + this.${fk?lower_case}.get${attribute.name?cap_first}() + "
                    </#if><#sep> , </@compress> " +
                    </#list>
                    <#if hk.foreignKeys?has_content>
                    <@compress single_line=true>", FK_${name?lower_case}Id = " +
get${name?cap_first}Id() + </@compress>
                    </#if>
                    " WHERE ${fk?lower_case}Id =" +
${fk?lower_case}.get${fk?cap_first}Id());
        </#list>

        sqLiteConn.close();
 }
</#list>
</#if>
```

Estes métodos apesar de haver foreign key vão fazer uma serie de verificações para que os acessos à base de dados sejam gerados corretamente. Assim o utilizador ao testar o ORM pode manipular com o apoio a estes métodos e facilitar a manipulação

Há que dar enfase também a um método genérico que é usado em todas as classes. Este método irá devolver uma lista com as informações sobre o objeto. Irá receber um caminho e uma query que vêem de outros métodos

gerados automaticamente e que o utilizador no ORM poderá usufruir para ir buscar informação à bd. O método em texto antes de ser gerado automaticamente é o seguinte:

```
private static ArrayList<${name}> get${name}ResultSet(String filename, String query)
throws SQLException {

   SQLiteConn sqLiteConn = new SQLiteConn(filename);
   ResultSet rs = sqLiteConn.executeQuery(query);
   ArrayList<${name}> ${name?lower_case}List = new ArrayList<>();

        if (rs.getMetaData().getColumnCount() == 0) {
            System.out.println("No Results");
            return null;
        }

        while (rs.next()) {
             ${name} ${name?lower_case} = new
${name}(rs.getInt(rs.getMetaData().getColumnName(1))<#if
attributes?has_content>, <#else>); </#if>
             <#if attributes?has_content>
             <#list attributes as attribute>
             <#compress><#if attribute.type ==
"String">rs.getString(rs.getMetaData().getColumnName(${attribute?index + 2}))
             <#elseif attribute.type ==
"int">rs.getInt(rs.getMetaData().getColumnName(${attribute?index + 2}))
             <#elseif attribute.type ==
"Date">rs.getDate(rs.getMetaData().getColumnName(${attribute?index + 2}))
             <#elseif attribute.type ==
"double">rs.getDouble(rs.getMetaData().getColumnName(${attribute?index +
2}))</#if></#compress><#sep>,
             </#list>);
             </#if>
    ${name?lower_case}List.add(${name?lower_case});
            }

   sqLiteConn.close();
   return ${name?lower_case}List;
}
```

Este método é privado pois a sua utilização é só na classe e o utilizador não poderá aceder ao mesmo diretamente para ir buscar uma lista de objetos.

2.6.2 sqlite3_create.ftl

Com a introdução de alguns extras como as relações, este template necessita de verificar as relações entre as classes. Quando o ficheiro de base de dados é criado com base nas relações poderá atribuir foreign keys às tabelas respetivas ou no caso de uma relação N para N irá criar uma nova tabela para se guardar as foreign keys de acesso.

2.6.3 xml_create.ftl

Este template permite que ao ser gerado um modelo novo seja gerado um ficheiro xml. Neste template é de notar que também será gerado mesmo se o modelo for criado a partir de um outro xml. No fim temos oportunidade de ter um ficheiro xml com as relações todas corrigidas.

2.7 Package utils

O Package utils guarda alguns packages necessários para as transformações e também construção dos ficheiros a partir de um modelo.

2.7.1 Package builder

O Package builder vai ser para guardar as classes que irão gerar ficheiros a partir de um modelo.

2.7.1.1 Builder.java

A class Builder é das mais importante do programa pois esta class vai tratar da criação dos modelos em ficheiros. Para além de criar os ficheiros vai também tratar das relações para que quando os ficheiros forem criados as relações estejam correctas. Para esta classe foram criados alguns métodos:

- **public void buildModel(Model model) throws IOException –** Este método é usado em várias partes do programa que envolve a criação dos ficheiros. E vai usar alguns métodos que estão incluidos nesta class. Inicialmente começa por obter as nossas tables.

- **public void createFile(File file, String sqlTables, Model model, Model2Text model2Text) –** Este método vai primeiro criar o nosso package com o nome do nome, depois vai criar a base de dados e de seguida as nossas classes, quando tiver as classes irá criar o respetivo ficheiro xml

- **public void createXML(Model model, Model2Text) throws IOException –** Este método vai adicionar ao package do modelo um novo package chamado XML, criar o respetivo xml do modelo e colocar nesta pasta.

- **public void createJavaFile(Class clazz, Model model, Model2Text model2Text) throws IOException –** Este método vai um ficheiro java com as informações da nossa classe. Através de uma transfomação model 2 text conseguimos gerar a string com a informação para colocar no ficheiro java.

- **public List<Class> makeRelations(Model model) –** Este método vai a cada classe e procurer na sua lista de relações que tipo de relações é que tem, se existirem. Quando encontra vai enviar para o método de cada respetivo tipo. No vim devolve uma lista de classes com as relações já tratadas. Este método é para ser usado antes do buildModel.

- **public void make1To1(List<Relations> relations, List<Class classList, int i, int j) –** Este método vai resolver as relações 1 para 1. Como tal como é uma relação 1 para 1 e encontramos uma relação, vamos adicionar à classe em que estamos a pesquisar as relações uma foreign key para essa relação. Dado que é uma relação 1 para 1 temos que também adicionar a relação na class com o nome da foreign key. Vamos então procurar na nossa lista de classes onde é que essa classe está. Se existir adicionamos uma foreign key e a cada uma dela uma hidden key com a informação da classe que é a foreign key.

- **public void make1ToN(List<Relation> relations, List<Class> classList, int i, int j) –** Método para resolver as relações do tipo 1 para n, Este método vai adicionar uma foreign key à classe com o nome igual ao da relação. Após adicionar uma relação vai adicionar às 2 uma hidden Key.

- **public void makeNToN(List<Relation> relations, List<Class> classList, Model model, int i, int j) –** Este método vai resolver as relações do tipo n para n. Inicialmente vamos verificar se a relação é possível. Para isto vamos procurar na nossa lista de classes se a classe com o nome da relação existe. Se encontrarmos adiciona-mos ao modelo uma relação com as duas classes. Este servirá para a criação das bases de dados da tabela extra e também poderá ser usado para as geração de código nas classes java. Depois de adicionarmos vamos a essa classe e apagar todas as relações que possam existir com a classe principal onde encontrámos a relação para que não haja duplicados. De seguida vamos verificar se a classe da relação tem uma foreign key para a classe principal. Caso tenha é removida. O mesmo é feito para a classe principal.

2.7.1.2 PreMade.java

A class PreMade tem como objetivo a criação de modelos. Para isso esta classe tem 2 métodos que irão gerar o modelo para o Person ou o BookStore. Estes modelos já têm as relações e estão prontos a ser construídos através do método buildModel da class Builder.

2.7.2 Package transformations

O Package tranformation trata das transformações, sejam estas Model 2 Model ou Model 2 Text. Para os extras foi necessário adicionar uns métodos à class M2M.

2.7.2.1 M2M.java

A class M2M foi criada para se guardar os métodos relacionados com a geração de modelos. Com o apoio da class Builder esta classe vai gerar os modelos a partir de um xml ou de um xmi. Para cada uma das situações foram criados dois métodos distintos que obtêm a informação de cada ficheiro individualmente quando necessário. Foi também adicionado um método para uma melhor organização de código que irá tratar dos atributos e das relações que as classes contêm.