Relatório DBM

MILESTONE 1 ORM

DBM – João Ventura

Tiago Monteiro 140221001

Relatório DBM

MILESTONE 1 ORM

Introdução

Este relatório tem como objetivo explicar o projeto desenvolvido para a CRUDApps para a geração de código automático, neste documento irão ser abordados temas como os modelos e meta-modelos utilizados, o workflow da aplicação, a arquitetura da aplicação e por fim os extras da aplicação que consistem em relações, herança, constrains, áreas protegidas, conexão MySql e uma pequena interface gráfica para ajudar na construção de modelos.

Contexto do projeto

O projeto consiste na aplicação dos conceitos aprendidos na cadeira de DBM para a geração de software orientada a modelos. O Projeto pretende desenvolver um software para a CRUDApps onde o processo de desenvolvimento de software era estático, optando-se então por se desenvolver um mecanismo de geração de código automática através de modelos.

Modelos e Meta-modelos

Neste projeto foi necessário criar modelos e meta modelos de modo a fazer transformação de M2M e M2T. Os meta-modelos criados são "Class", "Attribute" e "Relation" e "Model".

Class

A classe tem como objetivo representar uma Classe de uma linguagem com paradigma Objetos. Esta é definida por um conjunto de parâmetros essenciais para a geração do código.

Parâmetros

Name – O nome da classe

Attributes – Os atributos da classe (meta-modelo Atributo)

Relations – As relações que esta classe tem com outras classes

Package – O Package onde a classe se encontra inserida

Extend – Usado quando a classe herda de outra classe

SuperClass – Usando quando é superclasse de outra classe

ProtectedMethods – Os métodos protegidos que são para manter após nova geração

```
private String name;
private List<Attribute> attributes;
private List<Relation> relations;
private String pkg;
private Class extend;
private boolean superClass;
private String protectedMethods;
```

Figura 1 - Parâmetros da Classe

Attribute

Este meta-modelo serve para armazenar toda a informação sobre um atributo desde o sei nome e tipo ás suas *constrains*.

Parâmetros

Name – O nome do atributo

Type – O tipo do atributo

Required – Constrain required, se é um atributo que não pode ser nulo

Max – O valor máximo (Apenas para atributos numéricos)

Min – O valor mínimo (Apenas para atributos numéricos)

Contains – Se contem uma cadeia de caracteres (Apenas para String)

notEmpty – Se não é vazia (Apenas para String)

maxLength – O tamanho máximo da String

minLength – O tamanho mínimo da String

```
private String name;
private String type;
private boolean required;
private Integer max;
private Integer min;
private String contains;
private boolean notEmpty;
private Integer maxLength;
private Integer minLength;
```

Figura 2 - Parâmetros do meta-modelo Atributo

Relation

Este meta-modelo tem como objetivo representar as relações que podem existir entre as classes. Estas relações podem ser de 4 tipos sendo dois idênticos:

- OneToOne
- OneToMany
- ManyToOne
- ManyToMany

Parâmetros

Base – A classe base da relação

Target – A classe com quem a classe base tem uma relação

RelationType – Um dos 4 tipos de relação possíveis

```
private Class base;
private Class target;
private RelationType type;
```

Figura 3 - Parâmetros do meta-modelo Relation

Model

Este é o meta-model responsável por agrupar todos os outros, é para este Model que são feitas as transformações M2M. Este para além de ter como parâmetros os meta-modelos Class e Relation possui também o nome do modelo.

Workflow

Neste projeto foram encontradas três fases muito importantes, a transformação de M2M, a transformação de M2T a nível de geração de código de linguagens orientadas a objetos (ex: Java) e a fase de geração da base de dados também esta M2T que permite existir um ORM fazendo a ponte entre o código gerado em Java e os dados. De modo a que o projeto seja extensível e escalável estas três fases do workflow foram divididas em interfaces. Assim sempre que for necessária uma nova maneira de fazer uma das três fases basta criar uma nova classe que use a interface. Então o workflow para a geração de código passa por transformação M2M para obter o nosso modelo, depois a partir do modelo obtido faz-se uma transformação M2T para gerar o código para base de dados que é executado no momento e por fim é gerado código da linguagem de programação com o acesso á base de dados.

Build

O build consiste na junção das três fases mencionadas acima, uma parte importante neste build é a abstração criada devido ao uso de Interfaces, sendo estas:

M2M – Interface responsável pela transformação de um modelo no desejado

M2TClass – A Interface responsável por gerar código para paradigma Objetos

M2TBd – A interface responsável por gerar código para bases de dados

Assim para gerar o código só tem de se indicar classes que implementem estas interfaces, no exemplo da figura 4 temos como M2M o XMLToModel que converte xml no modelo, temos para M2TBd MySql que vai gerar código SQL apropriado para bases de dados MySql e temos como M2TClass Java que irá gerar código em java tendo noção qual a base de dados que usara para alimentar os objetos. Através desta solução é possível estender o programa para várias bases de dados ou várias linguagens sem comprometer nenhum dos outros processos.

```
Build build = new Build(new Java(),new XMLTOModel( filename: "src/model/person.xml"), new Mysql());
build.build();
```

Figura 4 - Código para fazer build

M2M

Esta fase corresponde á transformação de modelos importados para o modelo desejado que segue os meta-modelos criados neste projeto (página 2). Neste projeto é possível converter dois tipos de modelos para o modelo desejados, sendo estes XML e XMI. O XML terá de cumprir o DTD da figura-5 de modo a ser possível a sua transformação, um exemplo e um XML que cumpre este DTD é o XML da figura-6.

```
<?xml encoding="UTF-8"?>

        <!ELEMENT model (class*,relation*)>
        <!ATTLIST model
                name CDATA #REQUIRED>

        <!ELEMENT class (attribute)+>
        <!ATTLIST class
                name CDATA #REQUIRED
                extends CDATA #IMPLIED>

        <!ELEMENT relation EMPTY>
        <!ATTLIST relation
                base CDATA #REQUIRED
                target CDATA #REQUIRED
                type (ManyToMany|OneToMany|ManyToOne|OneToOne) #REQUIRED>

        <!ELEMENT attribute EMPTY>
        <!ATTLIST attribute
                name CDATA #REQUIRED
                type CDATA #REQUIRED
                required (true|false) #IMPLIED
                max CDATA #IMPLIED
                min CDATA #IMPLIED
                isAfter CDATA #IMPLIED
                isBefore CDATA #IMPLIED
                contains CDATA #IMPLIED
                notEmpty (true|false) #IMPLIED
                maxLength CDATA #IMPLIED
                minLength CDATA #IMPLIED>
```

Figura 5 - DTD para o XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE model SYSTEM "model.dtd">

<model name="Person">
    <class name="Super">
        <attribute name="ok" type="int" min="0" max="5"/>
        <attribute name="da" type="Date"/>
    </class>
    <class name="Person" extends="Super">
        <attribute name="name" type="String" required="true"/>
        <attribute name="age" type="int" required="true"/>
    </class>
    <class name="Manel" extends="Person">
        <attribute name="coi" type="String" contains="a" notEmpty="true" minLength="4" maxLength="9"/>
        <attribute name="man" type="int" />
    </class>
    <relation base="Person" target="Manel" type="ManyToMany"/>
</model>
```

Figura 6 - XML válido para o DTD criado

No projeto esta transformação é feita a partir de uma classe que implemente a interface M2M que possui o método getModel(). Para demonstração foram feitas duas conversões, a partir de xml e a partir de xmi, mas o programa não está limitado e pode-se expandir para outras conversões.

M2T

Esta é a fase do workflow onde é gerado código. Esta parte está dividida em geração de código orientado a objetos e geração de bases de dados.

Para gerar código orientado a objetos deve-se usar uma classe que implemente a interface M2TClass, pois é utilizada pelo build para gerar o código. Este código passa pela geração de atributos, gets e sets até ao código ORM (save, all, where, delete) e ainda código da parte das relações, sendo este bidirecional, ou seja, se a classe C1 tiver relação com a classe C2, através da classe C1 posso obter as instancias C2 com quem se relaciona C1 e a partir da classe C2 posso obter as instancias de C1 com quem C2 se relaciona. Um aspeto importante aqui é o tipo de base de dados que irá ser usado para alimentar os objetos com dados, por este motivo o método Convert() da interface M2TClass recebe como parâmetros o model e também uma classe da interface M2TBd, esta informação é inserida no template através de parâmetros partilhados como se pode observar na figura 7, sendo possível gerar o código usando das classes usando as mesmas propriedades partilhadas que irão ser usadas para incluir ficheiros específicos no template.

```
Model2Text model2Text = new Model2Text( pathname: "src/templates");
System.out.println(m2TBd.getBdTyp().toString());
model2Text.addShared("bdname", m2TBd.getBdTyp().toString());
model2Text.addShared("bdinclude",m2TBd.getIncludeClassFileName());
model2Text.addShared("bdincludefunctions",m2TBd.getImportFunctionsFileName());
```

Figura 7 - Parametros partilhados no template

Para gerar código que crie as bases de dados deve-se utilizar uma classe que implemente a interface M2TBd, pois é esta que contem informação importante como por exemplo o template a utilizar.

```
public class Mysql implements M2TBd {
    @Override
    public void convert(Model model) {
        Model2Text model2Text = new Model2Text( pathname: "src/templates");
        MysqlConnection sq = new MysqlConnection();
        String sqlTables = model2Text.render(model,  name: "mysql/mysql_create.ftl");
        System.out.println(sqlTables);
        sq.executeAll(sqlTables);
    }

    @Override
    public BD_Type getBdTyp() { return BD_Type.MySql; }

    @Override
    public String getIncludeClassFileName() { return "java_sqlite3.ftl"; }

    @Override
    public String getImportFunctionsFileName() { return "java_mysql_functions.ftl"; }
}
```

Figura 8 - Classe MySql que é responsável por gerar código SQL para bases de dados MySql

Arquitetura

Este projeto é dividido em 7 packages sendo estes M2M, M2T, M2TBd, metamodels, model, templates e utils.

M2M

Package responsável por guardar classes de transformações M2M

Classes

XMLToModel – Converte xml para Model

M2M (Interface) – Interface para conversores M2M como o XMLToModel

M2T

Package responsável por guardar classes de transformações M2T para linguagens de programação

Classes

M2TClass (Interface) – Interface para conversores M2T para linguagens de programação como java

Java – Converte o Model para código Java

M2TBd

Package responsável por guardar classes de transformações M2T para bases de dados

Classes

M2TBd (Interface) – Interface para conversores M2T para bases de dados

MySql – Classe responsável por gerar código para bases de dados MySql

Sqlite3 – Classe responsável por gerar código para bases de dados Sqlite3

Metamodels

Package responsável pelos meta-modelos

Model

Package responsável pelos modelos (xml, xmi)

Templates

Package responsável por guardar os templates

Pastas

Java – tempaltes de java

Sqlite3 – templates de sqlite3

MySql – templates de MySql

Utils

Package responsável por guardar utilitários tais como classe de acesso a bases de dados

Extras

Os extras que existem neste projeto são:

- Todos os tipos de relações
- Herança
- Constrains
- Áreas protegidas
- MySql
- Pequena interface

Em vez de ter sido apenas focado em apenas um extra como por exemplo uma interface muito boa com drag and drop, optou-se por mais extras que abrangem mais problemas a solucionar. Assim decidiu-se que neste projeto estes seriam os extras porque:

- É sempre preciso vários tipos de relações como por exemplo N-M,;
- Existem casos de herança como por exemplo um utilizador que tanto é estudante como professor, mas contem dados idênticos então a herança vai permitir uma programação mais próxima do paradigma objetos, mais próxima da implementação correta e ainda permitir com que o código pareça ainda mais que foi feito por um programador;
- É necessário validar dados como por exemplo idade, número de caracteres da password
- Quando se sincroniza o código gerado com o novo modelo podem existir métodos adicionados por um programador que quer que se mantenham então as áreas protegidas vão permitir isso
- Porque sqlite3 é para bases de dados locais e pequenas então pode existir necessidade de bases de dados relacionais mais elaboradas até porque no futuro este ORM irá ser consumido por um webservice e a quantidade de dados pode ser elevada.
- Porque facilita a construção e controlo dos modelos sem ser por xml

Relações

Foram implementados todos os tipos de relações e ainda de acesso bidirecional, ou seja, C1 aceder a C2 e C2 aceder a C1 quando existe uma relação entre C1 e C2.

OneToOne

Esta relação cria uma constrain unique de modo a ser uma relação única.

Exemplo:

No modelo xml temos a classe Aluno e a classe Curso, estabelecemos uma relação de uma para um para efeitos demonstrativos

```
<relation base="Aluno" target="Curso" type="OneToOne"/>
```

Na geração de código mysql é possível verificar a existência de duas constrains, uma de foreign key e outra de unique

```
ALTER TABLE Aluno ADD COLUMN `curso_id` INTEGER REFERENCES Curso (id);

ALTER TABLE Aluno ADD CONSTRAINT constraint_Curso UNIQUE (curso_id);
```

São criado ainda métodos adicionas nas classes java de acesso e adição de forma bidirecional

Classe curso

```
public Aluno getAluno() {


    String query = String.format("select * from Aluno  where curso_id ="+ this.id);
    ResultSet result = con.executeQuery(query);

        try {
            while (result.next()) {

                Aluno p = (Aluno) Aluno.getAlunoClass(result);
                return p;

            }

        } catch (SQLException e) {
            e.printStackTrace();
        }

    return null;

}
```

```
public void setAluno(Aluno aluno) {

    String query = String.format("Update %s set curso_id = '%d'  where id = %d",
"Aluno",this.id, aluno.getId());
    con.executeUpdate(query);

}
```

Classe Aluno

```
public Curso getCurso() {

    String query = String.format("select a.* from Curso as a inner join Aluno as b on a.id
= b.curso_id where b.id ="+ this.id);
    ResultSet result = con.executeQuery(query);

        try {
            while (result.next()) {

                Curso p = (Curso) Curso.getCursoClass(result);
                return p;
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }

    return null;

}

public void setCurso(Curso curso) {

    String query = String.format("Update %s set curso_id = '%d'  where id = %d", "Aluno",
curso.getId(), this.id);
    con.executeUpdate(query);

}
```

OneToMany e ManyToOne

Estas relações tem o mesmo funcionamento, neste caso o resultado é idêntico ao OneToOne mas a constrain única não é criada.

Exemplo:

No modelo xml temos a seguinte relação

```
<relation base="Aluno" target="Curso" type="OneToMany"/>
```

Podemos reparar que no código gerado a constrain única já não é gerada

```
ALTER TABLE Curso ADD COLUMN aluno_id INTEGER REFERENCES Aluno (id);
```

Os métodos são estes também idênticos

Classe curso

```
public Aluno getAluno() {
    // select * from Aluno as a inner join Curso as b on a.id = b.aluno_id where b.id =
this.id;

    String query = String.format("select a.* from Aluno as a inner join Curso as b on a.id
= b.aluno_id where b.id ="+ this.id);
    ResultSet result = con.executeQuery(query);

    try {
        while (result.next()) {

            Aluno p = (Aluno) Aluno.getAlunoClass(result);
            return p;
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return null;

}

public void setAluno(Aluno aluno) {

    String query = String.format("Update %s set aluno_id = '%d'  where id = %d", "Curso",
aluno.getId(), this.id);
    con.executeUpdate(query);

}
```

classe Aluno

```
public List<Curso> getCursos() {
    // select * from Person where manel_id = this.id;
    String query = String.format("select * from Curso where aluno_id = " + this.id);
    ResultSet result = con.executeQuery(query);
    List<Curso> lista = new ArrayList<>();

    try {

        while (result.next()) {

            Curso p = (Curso) Curso.getCursoClass(result);
            lista.add(p);

        }

        return lista;

    } catch (SQLException e) {
        e.printStackTrace();
        return null;
    }
}
```

ManyToMany

Neste tipo de relações é necessária uma tabela intermédia para permitir uma relação de muitos para muitos.

Exemplo

No modelo xml temos a seguinte relação

```
<relation base="Aluno" target="Curso" type="ManyToMany"/>
```

Podemos repara que o código MySql gerado apresenta ma tabela de relação

```
CREATE TABLE IF NOT EXISTS Relation_Aluno_Curso(
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   aluno_id INTEGER REFERENCES Aluno (id),
   curso_id INTEGER REFERENCES Curso (id)
);
```

As diferenças também se podem observar nas classes pois agora ambas retornam uma lista

Classe curso

```
public List<Aluno> getAlunos() {

    String query = String.format("select b.* from Relation_Aluno_Curso as a inner join
Aluno as b on a.aluno_id = b.id where a.curso_id = " + this.id);
    ResultSet result = con.executeQuery(query);
    List<Aluno> lista = new ArrayList<>();
        try {
            while (result.next()) {

                Aluno p = (Aluno) Aluno.getAlunoClass(result);
                lista.add(p);

            }
            return lista;

        } catch (SQLException e) {
            e.printStackTrace();
        }
    return null;
}
```

```
public void addAluno(Aluno aluno){

    String query = String.format("Insert into Relation_Aluno_Curso(aluno_id, curso_id)
values (%d, %d)",aluno.getId(),this.id);
    con.executeUpdate(query);

}
```

Classe Aluno

```
public List<Curso> getCursos() {

    String query = String.format("select b.* from Relation_Aluno_Curso as a inner join
Curso as b on a.curso_id = b.id where a.aluno_id = " + this.id);
    ResultSet result = con.executeQuery(query);
    List<Curso> lista = new ArrayList<>();
        try {
            while (result.next()) {

                Curso p = (Curso) Curso.getCursoClass(result);
                lista.add(p);
            }
            return lista;

        } catch (SQLException e) {
            e.printStackTrace();
        }
    return null;
}

public void addCurso(Curso curso){
    String query = String.format("Insert into Relation_Aluno_Curso(aluno_id, curso_id)
values (%d, %d)",this.id,curso.getId());

    con.executeUpdate(query);
}
```

Herança

Este extra corresponde á criação de herança entre classes, esta é importante para seguir o paradigma objetos, para uma arquitetura mais correta e para uma geração de código mais próxima de um código feito por um programador. Foi sugerido utilizar o tipo abstrato de dados para a herança mas este não pode ser usado em métodos estáticos do Java então continuou-se com a utilização do retorno de um objeto da superclass.

Exemplo

No xml criamos as classes e estabelecemos a herança com o extends

```
<model name="ESTS">
    <class name="Utilizador">
        <attribute name="nome" type="String" minLength="2" maxLength="255"/>
        <attribute name="dataNascimento" type="Date"/>
    </class>
    <class name="Aluno" extends="Utilizador">
        <attribute name="numero" type="int" required="true"/>
        <attribute name="anoInicio" type="Date" required="true"/>
    </class>
    <class name="Professor" extends="Utilizador">
        <attribute name="numero" type="int" />
        <attribute name="gabinete" type="String" />
    </class>
</model>
```

Podemos observar que são criadas constrains que indicam referencias dos id's das tabelas Aluno e Professor de modo a que estes para existirem o Utilizador tenha de existir

```
ALTER TABLE Aluno ADD CONSTRAINT fk_Utilizador_Aluno_id FOREIGN KEY (id)
REFERENCES Utilizador(id);

ALTER TABLE Professor ADD CONSTRAINT fk_Utilizador_Professor_id FOREIGN KEY (id)
REFERENCES Utilizador(id);
```

A nível das classes são provocadas algumas alterações a nível de encapsulamento e chamadas ao super no método save() e no de obter a classe bem como um construtor que constrói a partir da superclasse

```
public class Aluno extends Utilizador {
```

```
public Aluno(Utilizador e) {
    //Utilizador
    super();
    this.id = e.id;
    this.nome = e.nome;
    this.dataNascimento = e.dataNascimento;
}
```

```
@Override
public void save() {


    String query = "";

    if (this.id == -1) {

        super.save();
        query = String.format("Insert into %s(id, numero, anoInicio ) values (%d,
%d , '%s')", "Aluno",this.id, this.numero, new SimpleDateFormat("yyyy/MM/dd
HH:mm:ss").format(this.anoInicio) );
        con.executeUpdate(query);

    } else {

        query = String.format("Update %s set numero = %d , where id = %d",
"Aluno", this.numero, this.anoInicio, this.id);
        con.executeUpdate(query);

    }



}
```

```
public static Aluno getAlunoClass(ResultSet result) throws SQLException {

     int id = result.getInt("id");
     int numero = result.getInt("numero");
     Date anoInicio = result.getDate("anoInicio");

     //Manel p = new Manel(Person.get(id));
     Aluno p = new Aluno((Utilizador)Utilizador.get(id));
     p.setId(id);
     p.setNumero(numero);
     p.setAnoInicio(anoInicio);
    return p;

}
```

Constrains

As constrains são essenciais para garantir a validação de dados inseridos, de modo a não inventar a roda as constrains que existem neste projeto são baseadas em constrains encontradas noutros ORMs

Exemplo

Vamos utilizar um xml que utilize todas as constrains possíveis

```
<model name="ESTS">
    <class name="Utilizador">
        <attribute name="nome" type="String" required="true" contains="a"
notEmpty="true" minLength="2" maxLength="255"/>
        <attribute name="dataNascimento" type="Date"/>
    </class>
    <class name="Aluno" extends="Utilizador">
        <attribute name="numero" type="int" required="true" max="10000000"
min="1"/>
        <attribute name="anoInicio" type="Date" required="true"/>
    </class>
    <class name="Professor" extends="Utilizador">
        <attribute name="numero" type="int" />
        <attribute name="gabinete" type="String" />
    </class>
</model>
```

Podemos verificar a geração de verificações no código java

```
public void setNome(String nome) {

    if(!nome.contains("a")){
        throw  new RuntimeException("nome deve conter a string a");
    }

    if(nome.isEmpty()){
        throw  new RuntimeException("nome não pode ser vazio");
    }

    if(nome.length() > 255){
        throw  new RuntimeException("nome tem de ter menos de 256 carateres");
    }

    if(nome.length() < 2){
        throw  new RuntimeException("nome tem de ter mais de 1 carateres");
    }

    this.nome = nome;
}
```

```
public void setNumero(int numero) {

    if(numero > 10000000){
        throw  new RuntimeException("numero tem de ser inferior ou igual a 10,000,000");
    }

    if(numero < 1){
        throw  new RuntimeException("numero tem de ser maior ou igual a 1");
    }

    this.numero = numero;
}
```

vamos então correr o seguinte código exemplo para verificar se os erros são lançados

```
try {
    ut.setNome("");
} catch (Exception ex) {
    ex.printStackTrace();
}
try {
    ut.setNome("Igor");
} catch (Exception ex) {
    ex.printStackTrace();
}
try {
    ut.setNome("a");
} catch (Exception ex) {
    ex.printStackTrace();
}

try {

ut.setNome("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
+

"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +

"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +

"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
} catch (Exception ex) {
    ex.printStackTrace();
}
```

```
java.lang.Exception: nome não pode ser vazio
    at proj.ests.Utilizador.setNome(Utilizador.java:43)
    at Main.main(Main.java:60) <5 internal calls>
java.lang.Exception: nome deve conter a string a
    at proj.ests.Utilizador.setNome(Utilizador.java:47)
    at Main.main(Main.java:65) <5 internal calls>
java.lang.Exception: nome tem de ter mais de 1 carateres
    at proj.ests.Utilizador.setNome(Utilizador.java:55)
    at Main.main(Main.java:70) <5 internal calls>
java.lang.Exception: nome tem de ter menos de 256 carateres
    at proj.ests.Utilizador.setNome(Utilizador.java:51)
    at Main.main(Main.java:76) <5 internal calls>
```

Áreas protegidas

As áreas protegidas têm um papel importante na sincronização de código já gerado, pode acontecer apos uma primeira geração de código o programador adicionar métodos ás classes que o ORM não gerava, ao sincronizar o programador não quer perder esses métodos, mas que que as classes sofram uma sincronização. Assim implementou-se uma estratégia através de comentários como irá ser demonstrado no exemplo. Para a criação destas áreas são precisos os comentários

```
//@Protected Start
```
Método

```
//@Protected End
```

Exemplo

Criamos duas áreas protegidas na classe Aluno

```
//@Protected Start
 public void teste1(){
 }
//@Protected End

 // @Protected Start
 public void teste2(){
 }
//@Protected End
```

Gerou-se o novo código e pode-se verificar que os métodos manterem-se e continuam protegidos

```
//@Protected Start
public void teste1(){
}

public void teste2(){
}

//@Protected End
```

Mysql

O objetivo da geração de mysql foi de testar as Interfaces criadas para verificar se o projeto era escalável e também porque até agora em laboratórios só se tinha gerado código para Sqlite3 mas caso a aplicação CRUD precise de uma base dados com mais funcionalidades e melhora para armazenar grandes quantidades de dados.

Para que isto fosse possível criou-se uma importação dinâmica nos templates usando parâmetros shared do Freemaker como no exemplo abaixo

```
Model2Text model2Text = new Model2Text("src/templates");
model2Text.addShared("bdname", m2TBd.getBdTyp().toString());
model2Text.addShared("bdinclude",m2TBd.getIncludeClassFileName());
model2Text.addShared("bdincludefunctions",m2TBd.getImportFunctionsFileName());
```

Utilizando estes parâmetros faz-se uma dinâmica no template de java

```
<#import bdincludefunctions as bd >
```

Foram criadas funções nesta classe tanto para sqlite3 como para mysql que irão ser usadas pelo template java

Para mysql

```
<#macro bd_imports >
import utils.sqlite.MysqlConnection;
import utils.sqlite.BdConnection;
</#macro>

<#macro bd_instance >
MysqlConnection.getInstace();
</#macro>
```

Para sqlite3

```
<#macro bd_imports >
import utils.sqlite.BdConnection;
import utils.sqlite.SQLiteConn;
</#macro>

<#macro bd_instance >
SQLiteConn.getInstace("src/proj/ORM.db");
</#macro>
```

Estão são usadas ao longo do template java para criar a dinâmica como por exemplo nos imports

```
<@bd.bd_imports />
```

Interface gráfica

De modo a facilitar o controlo do modelo e alterações sem ser por xml foi criada uma interface simples com as seguintes funcionalidades

- Adicionar/Remover classe
- Adicionar/Remover relação
- Adicionar/Remover Atributo
- Editar constrains
- Importar xml/xmi
- Fazer build

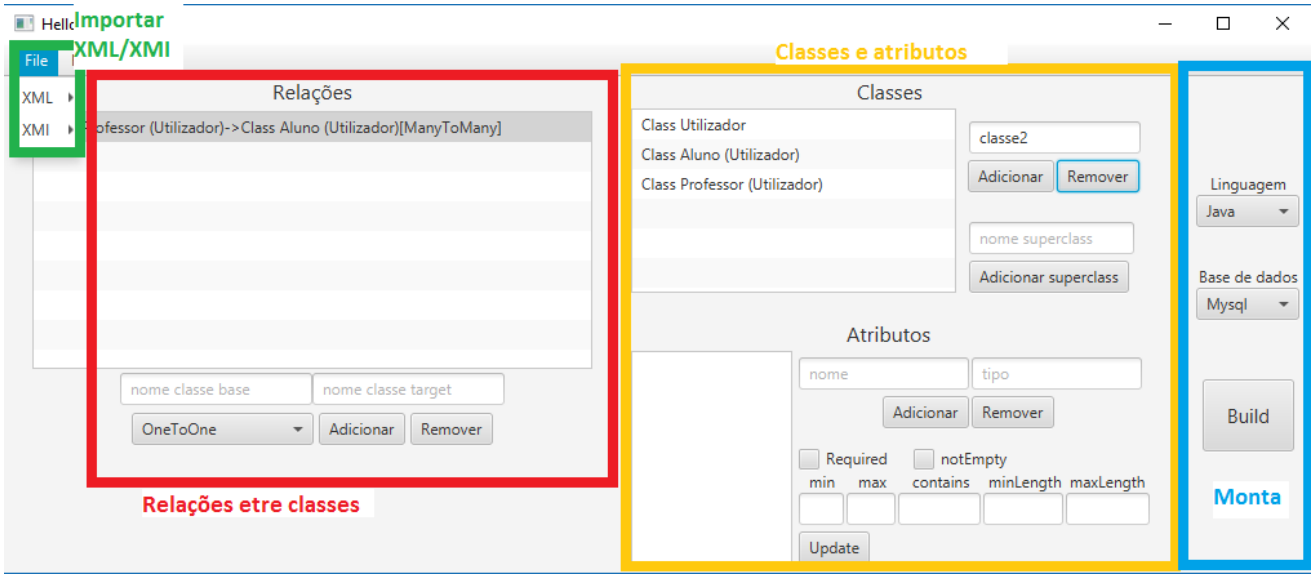

Figura 9 - Interface Gráfica

Conslusão

Com o desenvolvimento deste projeto e com os conceitos adquiridos na cadeira de DB conclui-se que a geração de código cada vez tem mais impacto na criação de softwares, os próprios IDE's geram código e diria que cada vez mais a geração de código estará mais presente devido ao aumento de abstração e á necessidade de software rápido.