FUNDAÇÃO EDUCACIONAL DO MUNICÍPIO DE ASSIS
INSTITUTO MUNICIPAL DE ENSINO SUPERIOR DE ASSIS
Comissão do PIC-IMESA

O uso de programação reflexiva para o desenvolvimento de aplicações comerciais adaptativas

Bolsista: Jefferson Simão Gonçalves
Orientador: Dr. Almir Rogério Camolesi
Coordenadoria de Informática IMESA – FEMA - Assis
Área de Pesquisa: Ciência, Tecnologia e Educação.

FICHA CATALOGRÁFICA

GONÇALVES, Jefferson Simão.

O uso de programação Reflexiva para o desenvolvimento de aplicações comerciais adaptativas. / Jefferson Simão Gonçalves

Fundação Educacional do Município de Assis – FEMA – Assis, 2012.
Páginas

Orientador: Dr. Almir Rogério Camolesi
Projeto de Iniciação Científica - Instituto Municipal de Ensino Superior de Assis – IMESA.

Sumário

Lista de Figuras

Lista de Códigos

1. Introdução

Quando se fala em sistemas, logo se vem a mente sistemas de computadores que automatizam tarefas diárias de uma empresa, mas sua atuação é mais ampla, para acompanharmos esta evolução é necessário entender a evolução do desenvolvimento cientifico e da inteligência humana. Além do desenvolvimento cientifico, parâmetros contidos em problemas crescem de uma forma assustadora e cada vez mais complexa. Impossibilitando que apenas uma área compreenda todas as informações e explicações destes fenômenos existentes.

Segundo Yourdon (1990) sistemas pode ser o conjunto estruturado ou ordenado de partes de elementos que se completam ou se mantêm em interação, ou seja, em operação recíproca na busca da consecução de um ou vários objetivos. De um modo mais simples podemos definir sistemas como conjuntos de elementos interdependentes, ou partes que interagem formando um todo unitário e complexo. Pode se afirmar ainda que um sistema possui partes divididas como sistemas menores que fazem parte de um sistema maior.

Sistemas devem ser construídos para atender as demandas das empresas. Neste contexto os desenvolvedores de softwares a cada dia mais tem se deparado com sistemas complexo e que tem que ser adaptados às exigências mais fundamentais segundo as características de cada empresa.

Aplicações complexas são caracterizadas por componentes e aspectos cuja estrutura e comportamento, comumente, podem modificar-se (CAMOLESI, 2004). Tais aplicações possuem um comportamento inicial definido por um conjunto de ações que desempenham suas funções elementares, e durante a execução podem ter o seu comportamento modificado para dar suporte a novas funcionalidades. Tais modificações são decorrentes dos estímulos de entrada a que são submetidos no sistema e/ou da ocorrência de suas ações internas.

Uma técnica utilizada para auxiliar os projetistas na modelagem de aplicações com comportamento modificável é a tecnologia adaptativa (NETO, 1993). A tecnologia adaptativa envolve um dispositivo não-adaptativo (subjacente) já existente em uma camada adaptativa que permite realizar mudanças no comportamento da aplicação definida (PISTORI , 2003).

Estudos já foram realizados com o objetivo de implementação de aplicações adaptativas[1]. Casachi (2011) realizou um estudo que teve por objetivo apresentar o uso da Programação Orientada a Aspectos (KICZALES, 1997) no desenvolvimento de aplicações que utilizam os conceitos de Tecnologia Adaptativa. Tal estudo permitiu a implantação da Tecnologia Adaptativa de uma forma fácil e segura, além de permitir que novas funcionalidades (aspectos) sejam adicionadas ao software sem a necessidade de modificar o código fonte já produzido. O programador deve apenas acrescentar novos aspectos ao software e o mesmo se adapta ao código já existente. Porém neste trabalho verificou-se que os conceitos de Tecnologia Adaptativa não são implementados completamente, uma vez que os programas que utilizam Programação Orientada a Aspectos são combinados antes da compilação e os programas produzidos não acabam se tornando adaptativos em tempo de execução. Neste trabalho é sugerido que estudos sejam realizados com o foco no desenvolvimento de aplicações que permitam a mudança de comportamento em tempo de execução utilizando-se dos conceitos de Programação Reflexiva.

Os programas geralmente são escritos para trabalhar com dados. Eles em geral, lêem, escrevem, manipulam e exibem dados (a geração de Gráficos a partir de bases de dados pré-existentes são um exemplo típico de programas desse tipo). Os tipos que você, como o programador, criar e usar são projetados para estes fins, e é você, em tempo de projeto, que deve compreender as características dos tipos que você usa. Para alguns tipos de programas, no entanto, os dados que eles manipulam não são números, textos ou gráficos, mas são as informações sobre programas e tipos de programa. Tais informações são conhecidas como metadados (Uma informação sobre a informação). Os Atributos são um mecanismo para a adição de metadados, tais como instruções do compilador e outros dados sobre seus dados, métodos e classes, para o próprio programa. Um programa pode olhar para os metadados de outros conjuntos ou de si mesmo, enquanto ele está executando. Quando um programa em execução olha para seus próprios metadados, ou de outros programas, a ação é chamada de Reflexão (*Reflection***)** (Lecture Notes, 2000).

Dessa forma, Reflexão é o processo pelo qual um programa pode ler seus próprios metadados. Um programa que faz Reflexão sobre si mesmo, extrai metadados de sua montagem e usa estes metadados ou para informar ao usuário ou para modificar o seu próprio comportamento.

[1] http://www.pcs.usp.br/~lta/

Reflexão é o conceito que tem a capacidade de ler metadados em tempo de execução. Usando a reflexão (*Reflection*), é possível descobrir os métodos, propriedades e eventos de um tipo, e então invocá-los dinamicamente, além de permitir que novos tipos sejam criados em tempo de execução (Lecture Notes, 1999).

Motivações para Reflexão Computacional

Os sistemas de informação atuais existentes no mercado apresentam um comportamento estático com estruturas predefinidas, nos quais são informados apenas os seus parâmetros inicias e estes são utilizados durante o tempo de execução. O mundo empresarial é dinâmico, novas funcionalidades e estratégias de cálculos surgem no mundo dos negócios o que acarretam em mudanças que devem ser realizadas nos sistemas existentes. Devido o comportamento de tais sistemas ser estático torna impossível, na maioria dos casos, mudanças automáticas ou sem a necessidade de recodificação de trechos e partes do sistema já desenvolvido. Tais mudanças além de demandar tempo e a alocação de profissionais para a sua realização, pode gerar problemas imprevistos nas aplicações e, até mesmo, terem altos custos para serem realizadas. O uso da Tecnologia Adaptativa neste tipo de aplicação tem por objetivo a obtenção de uma aplicação que possa ter seu comportamento alterado com a obtenção de novas informações adquiridas durante a execução da aplicação.

Neste contexto estudos serão realizados com o objetivo de aplicar técnicas de programação que permitam a construção de aplicações que possibilitem a mudança de comportamento em tempo de execução.

Objetivos do Trabalho

Realizar o estudo dos conceitos de Programação usando Reflexão e aplicar os conceitos obtidos no desenvolvimento de aplicações comerciais que possuem características adaptativas.

Objetivos Específicos

O objetivo desse trabalho foi estudar o conceito de desenvolvimento de aplicações usando Reflexão e a utilização destes conceitos no desenvolvimento de aplicações comerciais usando conceitos de tecnologia adaptativa.

Para que os objetivos delineados acima fossem atingidos alguns estudos específicos foram abordados:

- estudo da linguagem de programação JAVA que implemente o conceito de Reflexão;
- estudo dos conceitos de aplicações comerciais que permitam mudança de comportamento;
- definição de um estudo de caso que permite representar os conceitos estudados;
- implementação do estudo de caso definido;

Justificativa

Mesmo não sendo um assunto novo, existem poucos estudos relacionados ao uso da tecnologia adaptativa no desenvolvimento de sistemas de informação, principalmente, na área de softwares comerciais. Esse trabalho pretende mostrar o uso da tecnologia adaptativa no projeto e desenvolvimento de softwares comerciais, e como os softwares produzidos com conceitos desta tecnologia poderão ajudar os desenvolvedores de software e os seus respectivos usuários.

Estrutura do trabalho

Este trabalho encontra-se organizado da seguinte forma: inicialmente, no Capítulo 1 foram apresentados uma visão geral do trabalho, seus objetivos e justificativas. Na sequencia, o Capitulo 2 apresenta uma breve revisão bibliográfica dos conceitos de reflexão computacional e sua forma de utilização com base na linguagem JAVA. Com base nos estudos realizados é apresentado um estudo de caso, no Capítulo 3, que foca no projeto e implementação de um framework para geração automática de aplicações comerciais utilizando-se de técnicas de reflexão computacional. Por fim, no capítulo 4 são apresentadas algumas conclusões e trabalhos futuros.

2. Revisão Bibliográfica

No desenvolvimento de sistemas comerciais, o analista vai programá-lo para solucionar problemas, independente da quantidade de problemas. Esse sistema jamais vai modificar seu comportamento ou estrutura para solucionar um problema, pois ele irá realizar tudo o que foi programado sem fazer alterações em si mesmo, não permitindo que o sistema consiga se autoanalisar conforme novas necessidades que aparecerem.

Porém ao utilizar reflexão computacional é possível o sistema raciocinar sobre si mesmo, podendo solucionar problemas em tempo de execução, os quais somente poderiam ser resolvidos com adição de novos comandos na programação.

Segundo Barth (2000), o conceito básico do paradigma de reflexão computacional não é exatamente novo. Este conceito originou-se em lógica matemática e recentemente, mecanismos de alto nível tornam o esquema de reflexão um aliado de características operacionais ou não funcionais a módulos já existentes.

A reflexão pode ser definida como qualquer ação executada por um sistema computacional sobre si próprio, onde tal conceito foi apresentado pela primeira vez por Maes (1987), pode-se dizer que reflexão é a capacidade de um programa reconhecer detalhes internos em tempo de execução que não estavam disponíveis no momento da compilação do programa.

Reflexão Computacional tem dois significados distintos. Um é introspecção, que se refere ao ato de examinar a si próprio, o segundo é intercessão, onde está ligado ao redirecionamento da luz, ou seja, o termo reflexão computacional na área da informática tem a capacidade de examinar ou conhecer a si mesmo ou estrutura, podendo fazer alterações no seu comportamento através do redirecionamento ou de interceptação de operações efetuadas [Conceitos Básicos].

Segundo Barth (2000), a reflexão Computacional define em uma nova arquitetura de software. Este modelo de arquitetura é composto por um meta-nível, onde se encontram estruturas de dados e ações a serem realizadas sobre o sistema objeto, localizadas no nível base. Neste contexto, a arquitetura de meta-nível é explorada para expressar propriedades não funcionais do sistema, de forma independente do domínio da aplicação.

Requisitos funcionais não apresentam nenhuma função a ser realizada pelo software, e sim comportamentos que este software deve utilizar.

A Figura 1 ilustra a visão de um processo de reflexão em sistema computacional, o sistema pode ser dividido em vários níveis, o usuário envia uma informação ao sistema ou mensagem, e tratado pelo nível funcional, que é capacitado para executar a tarefa, assim o nível não funcional e responsável por realizar o gerenciamento do nível funcional.

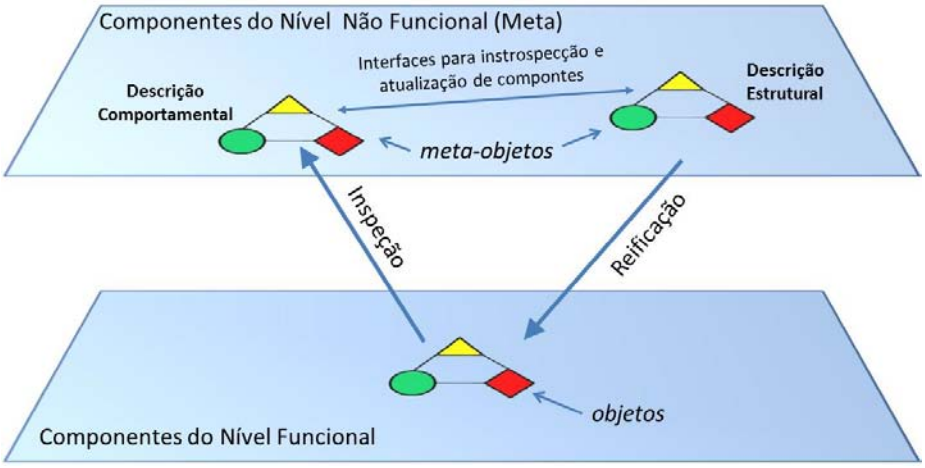


Figura 1. Arquitetura Geral de um Sistema Reflexivo.

Modelos de Reflexão

A dois modelos de Reflexão Computacional, o primeiro é a reflexão estrutural, pode ser definida por qualquer atividade exercida em uma metaclasse, a reflexão estrutural permite em tempo de execução fazer alterações em seus componentes no processo de execução ou compilação (SALLEM, 2007).

Reflexão estrutural tem as funcionalidades de realizar algumas transformações básicas, tais como: informações sobre a classe, suas instâncias, modificar atributos e métodos de uma classe, alterar campos (BARTH, 2000).

A Segunda Reflexão seria a reflexão comportamental, que leva mais a fundo sobre o comportamento de objetos, sem modificar a estrutura do objeto.

Arquitetura Reflexiva

A reflexão Computacional define conceitualmente uma arquitetura em níveis, denominada arquitetura reflexiva. Em uma arquitetura reflexiva, um sistema computacional é visto como incorporando dois componentes: um representando os objetos (domínio da aplicação), e outro a parte reflexiva (domínio reflexivo ou autodomínio) [Arquitetura Reflexiva].

No nível-base são encontradas as funções dos objetos, essas computações dos objetos têm a funcionalidade de resolver problemas e retornar informações sobre o domínio externo, enquanto o nível reflexivo encontra-se no meta-nível, resolvendo os problemas do nível-base, e retorna informações sobre a computação do objeto, podendo adicionar funcionalidades extras a este objeto.

Em uma arquitetura reflexiva, tem uma visão que um sistema computacional tem incorporado uma parte objeto e outra parte reflexiva. Na parte da computação objeto é realizadas funções para resolver problemas e retornar informações sobre um domínio externo, e quanto ao nível reflexivo tem a funcionalidade de resolver problemas e retornar informação sobre a computação do objeto.

Reflexão em Tecnologia JAVA

A API Java Reflection representa classes, interfaces e objetos presentes na JAVA Virtual Machine (JVM). Reflexão na tecnologia Java é um conceito muito poderoso e pode ser muito útil. Por exemplo, mapear objetos de uma classe gerando instruções SQL (update, delete, insert) referente às tabelas no banco de dados em tempo de execução [Tutorial Java Reflection].

Segundo Sousa, (2002), Reflection na tecnologia Java é constituída dos pacotes java.lang.reflect e Java.lang.string, e permite mudar o comportamento do programa, podendo fazer inspeções sobre definições da classe.

Classes Básicas

A seguir será apresentado definições de algumas classes existentes na tecnologia Java:

- JAVA.LANG.REFLECT: pacote que contém as classes básicas, introduzido na release JDK 1.1.;
- JAVA.LANG.CLASS: representa Classes e Interfaces em um aplicativo Java em execução. Class não tem nenhum construtor público, os objetos Class são construídos automaticamente pela JVM conforme as classes são carregadas;
- JAVA.LANG.PACKAGE: provê Informações sobre um pacote;
- JAVA.LANG.CLASSLOADER: provê informações sobre classes abstratas e operações para carregamento de classes;

- JAVA.LANG.REFLECT.MEMBER: interface que identifica informações sobre membros simples, atributos, métodos ou um construtor. Fornece informações e acesso a um único método em uma classe ou interface. O método refletido pode ser um método de classe ou um método de instancia (incluindo um método abstrato);
- JAVA.LANG.REFLECT.MODIFIER: provê métodos estáticos e constantes para decodificar modificadores de acesso de classe e membros;
- JAVA.LANG.REFLECT.ARRAY: disponibiliza métodos estáticos para criação e acesso a arranjos (arrays) Java;
- JAVA.LANG.REFLECT.CONSTRUCTOR: fornece informações para acessar os construtores de uma classe;
- JAVA.LANG.REFLECT.FIELD: utilizada para obter informações para acessar os campos de uma classe – campos de instancia de classe ou interface;
- JAVA.LANG.REFLECT.METHOD: Prover informa-ções para acessar os métodos de uma classe ou interface;
- JAVA.LANG.REFLECT.PROXY: adicionada na release JDK 1.3, com objetivo de prover métodos estáticos para criação de proxies de classes dinamicamente;
- JAVA.LANG.REFLECT.INVOCATIONHANDLER:Interface implementada pela instância de um Proxy.

3. *Framework* para Geração de Aplicações Comerciais

Com o foco no estudo e aplicação dos conceitos de reflexão computacional foi definido um estudo de caso que consiste no projeto e implementação de um *framework* para geração de código automático para a criação de sistemas de gestão comercial. O *framework* concebido tem por objetivo de utilizar as definições de um banco de dados existente, gerar automaticamente as classes para configuração, persistência de banco de dados e, posteriormente, as interfaces gráficas para o usuário. Tais interfaces gráficas permitirão que um usuário realize manutenções dos dados nas tabelas de banco de dados existente. Um programador ao iniciar um novo projeto gasta grande parte de seu trabalho gerando códigos para realizar tais tarefas, desta forma o *framework* gerado permitirá uma maior produtividade aos desenvolvedores de software.

Projeto do Framework para geração automática de código e interfaces gráficas

O *framework* proposto foi estruturado de forma que o mesmo gere os códigos e interfaces gráficas a partir da definição de um modelo lógico de banco de dados construído previamente por projetista de banco de dados. O *framework* também deve gerar um código organizado em camadas, seguindo os preceitos da programação orientada a objetos em camadas. Desta forma o código gerado será organizado e permitirá, conforme necessidade de um programador, que o mesmo possa dar manutenção no código, adicionando novas funcionalidades de uma forma fácil, eficiente e com boa produtividade.

O *framework* concebido também levou em consideração a geração de interfaces gráficas simples e objetivas. Para facilitar futuras necessidades de trocas de bases de dados e/ou novas implementações também foi definido que o mesmo deveria ter uma camada de persistência a dados que faz uso de outros *frameworks* de persistência a dados já existentes.

Com base nos requisitos mínimos definidos para a concepção do framework o mesmo foi organizado em cinco etapas que permitem desde a configuração do ambiente até a geração do código e aplicação de forma automática. As etapas são encadeadas tendo como entrada um determinado banco de dados como saída à aplicação comercial gerada. A Figura 2 ilustra as etapas e o funcionamento do *framework* definido.

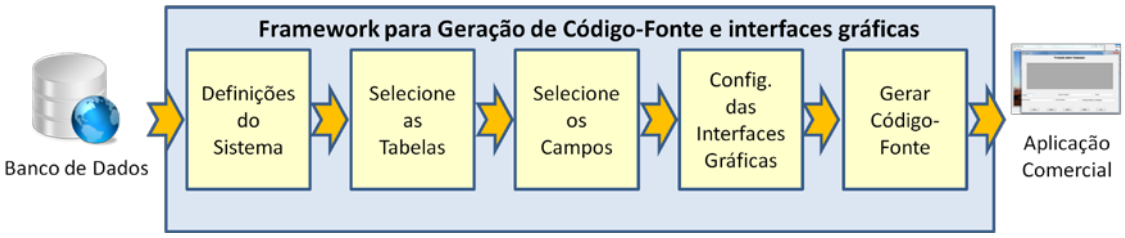


Figura 2. Etapas do Framework geração automática de código.

Conforme apresentado na Figura 2 o framework é definido em cinco etapas sendo:

- **Definições do sistema**: etapa responsável por receber como entrada a definição do banco de dados, realizar as suas configurações e escolha de uma determinada base, para geração de uma aplicação comercial;
- **Selecione as tabelas**: depois de definidas as configurações do banco de dados é apresentado ao usuário a descrição das tabelas da base de dados escolhida. Nesta etapa o usuário poderá escolher quais tabelas deseja gerar códigos para persistência dos dados e interfaces gráficas para manipulação dos dados pertinentes a tabela;
- **Selecione os campos**: permite ao usuário escolher quais campos de cada uma das tabelas definidas na fase anterior comporão as interfaces gráficas;
- **Configuração das Interfaces Gráficas**: Com base nos campos escolhidos o usuário poderá informar características para cada um dos campos, de forma que as interfaces gráficas geradas manipulem os dados de forma correta;
- **Gerar Código-Fonte**: por fim, depois de terem sido definidos para uma determinada base de dados quais são as tabelas e suas características é gerado o código-fonte de forma automática, produzindo como saída uma aplicação comercial.

Implementação do *framework.*

Na etapa inicial Definições do Sistema, o usuário ao ter acesso a interface gráfica do *framework* poderá selecionar qual o Sistema Gerenciador de Banco de Dados (SGBD) ele deseja utilizar. Nesse processo, depois de ser selecionado o SGBD, o *framework* se encarrega de obter as informações necessárias, a partir das *interfaces* das classes definidas, utilizando-se para isto algumas técnicas de reflexão.

A Figura 3 apresenta a interface do usuário que é oferecida para que o mesmo realize as informações de Banco de Dados necessárias para a aplicação que será definida.

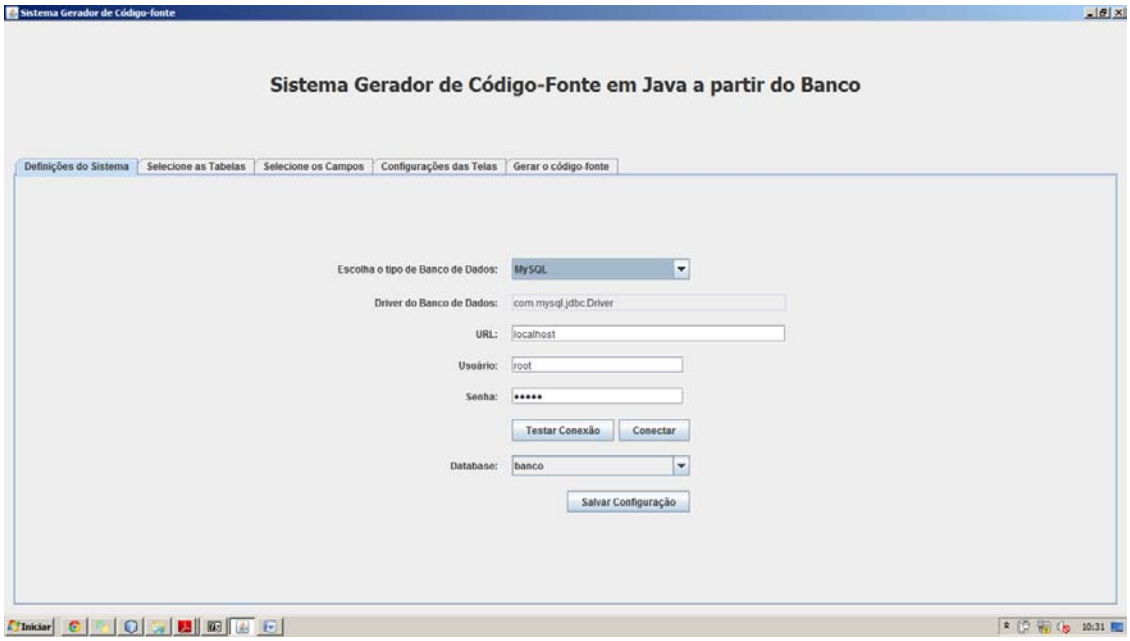


Figura 3. Interface para informação de configurações de Banco de Dados.

No trecho de Código 1,apresentado abaixo, faz o uso da classe *ReflectionUtils* responsável por manipular as informações de banco de dados definidas na interface do usuário e realizar as configurações de acesso a dados necessárias a nova aplicação que está sendo desenvolvida. O código consiste de um método de ação da caixa de seleção do Banco de Dados. Tal código é responsável por recuperar o valor selecionado na caixa de combinação definido na interface de usuário apresentada na Figura 3. O código faz a chamada ao método *carregarClasse*, local aonde o *framework* vai carregar em tempo de execução a *interface* referente ao SGBD selecionado. Utilizando o método *Class.forName* o *framework* monta o caminho da interface para definir a classe responsável pelo acesso aos dados.

Código 1. Trecho de programa para obter configurações de Banco de Dados.

```
private void cmbBancoDadosActionPerformed(java.awt.event.ActionEvent evt) {
    if (cmbBancoDados.getSelectedIndex() != -1) {
        confBanco.setTipoBanco(cmbBancoDados.getSelectedItem().toString());
        carregarClasse();
        txtDriverBanco.setText(
                ReflectionUtils.getValorCampo(bancoConfig, "DRIVER", new Object()).toString());
        confBanco.setClasse(
                ReflectionUtils.getValorCampo(bancoConfig, "DIALECT_HIBERNATE", new Object()).toString());
        urlBanco = ReflectionUtils.getValorCampo(bancoConfig, "URL", new Object()).toString();
    }
}
```

```
private void carregarClasse() {
    try {
        bancoConfig = Class.forName(Constantes.PACOTECONSTANTES + confBanco.getTipoBanco() + "Constantes");
    } catch (ClassNotFoundException ex) {
        throw new ClasseNaoEncontradaExcecao(ex);
    }
}
```

Para exemplificar o uso do recurso definido anteriormente, supõe-se o carregamento do banco de dados MySql. Inicialmente o *framework* vai montar o caminho da classe ficando da seguinte forma:

br.edu.fema.interfaces.constantes.banco.MySQLConstantes

Em seguida, após o carregamento da classe responsável pelo gerenciamento do banco de dados é carregada a classe responsável por obter algumas informações pertinentes ao funcionamento do SGBD selecionado, por exemplo, o driver do banco de dados, para evitar que o usuário tenha de ficar informando o mesmo toda vez que acessar a aplicação.

Outro exemplo da utilização do conceito de reflexão no trecho Código 2 é o método responsável pela obtenção de um determinado campo que esta definido na classe *ReflectionUtils*. Para este método são passados três argumentos o nome da classe, o campo e um objeto do tipo *object*.

Código 2. Classe *ReflectionUtils*.

```
public class ReflectionUtils {

    public static Object getValorCampo(Class<?> cls, String campo, Object object) {
        try {
            return cls.getDeclaredField(campo).get(object);
        } catch (IllegalArgumentException ex) {
            throw new ArgumentoIlegalExcecao(ex);
        } catch (IllegalAccessException ex) {
            throw new AcessoIlegalExcecao(ex);
        } catch (NoSuchFieldException ex) {
            throw new CampoNaoEncontradoExcecao(ex);
        } catch (SecurityException ex) {
            throw new SegurancaExcecao(ex);
        }
    }
```

Supondo a chamada do método *ReflectionUtils*, como retorno do método *System.out.println* tem-se:

System.out.println(ReflectionUtils.getValorCampo(MySQLConstantes.class,"DRIVER", new Object()).toString());

Com a chamada do respectivo código de programa obtém os detalhes referente ao driver do banco de dados que será manipulado, neste caso, o MySql, sendo retornado então com.mysql.jdbc.Driver . Depois de carregado o driver e a interface referente ao SGBD, o usuário poderá informar o endereço (url) do banco de dados, que consiste no endereço do servidor (host) mais a porta de conexão (*socket*).

Supondo a existência de um SGBD em uma máquina local, tem-se a url:

localhost:3306

No caso de ser um servidor que se localiza na internet, o usuário poderia colocar o IP, seguido da porta ou o nome do servidor dns, por exemplo, (mysql.fema.edu.br:3306) seguido da porta em que o banco de dados responde, no caso do exemplo (3306) utilizada pelo MySQL.

Na sequencia deve informar o usuário e a senha do SGBD. Depois de informar esses campos o usuário poderá testar a conexão para verificar se está tudo certo.

Testando a conexão

Com o objetivo de ilustrar a forma como ocorre a conexão com o banco de dados é apresentado no Código 3 o trecho de código utilizado no pressionamento de um botão. Tal código ilustra o carregamento dos valores que o usuário informou na interface de usuário definido na Figura 3 e faz um chamada ao método *TestarConexao* da classe *TestarConexao*, responsável por realizar a conexão com o banco de dados e utilizar os métodos de acesso de banco de dados definidos anteriormente.

Código 3. Trecho de Código para testar a conexão.

```
771
772     private void btnTestarConexaoActionPerformed(java.awt.event.ActionEvent evt) {
773         confBanco.setDriver(txtDriverBanco.getText());
774         confBanco.setUrl(txtUrl.getText().toString());
775         confBanco.setUrlCompleto(urlBanco + txtUrl.getText().toString());
776         confBanco.setUsuario(txtUsuario.getText().toString());
777         confBanco.setSenha(txtSenha.getText().toString());
778         boolean resultado = false;
779         resultado = TestarConexao.TestarConexao(confBanco);
780         if (resultado) {
781             JOptionPane.showMessageDialog(rootPane, "Conexão feita com sucesso!!");
782         } else {
783             JOptionPane.showMessageDialog(rootPane, "Erro na conexão!!");
784         }
785     }
786
```

O método *TestarConexao* definido no Código 4 é responsável por realizar um teste para verificar se consegue estabelecer uma conexão com o banco de dados informado. Se o teste de conexão for efetuado com sucesso. O usuário poderá prosseguir selecionando a opção *Conectar*.

Código 4. Método *TestarConexao*.

```
9   public class TestarConexao {
10
11      public static boolean TestarConexao(ConfiguracaoBE confBanco) {
12          try {
13              Class.forName(confBanco.getDriver());
14              DriverManager.getConnection(
15                      confBanco.getUrlCompleto(), confBanco.getUsuario(), confBanco.getSenha());
16              return true;
17          } catch (SQLException ex) {
18              throw new SQLExcecao(ex);
19          } catch (ClassNotFoundException ex) {
20              throw new ClasseNaoEncontradaExcecao(ex);
21          }
22      }
23  }
```

Ao escolher a opção *Conectar* (serão carregadas todas as informações para um objeto necessárias para ser realizado uma operação de seleção (*Select*) no banco de dados.

Código 5. Trecho de código para Conectar ao Banco de Dados.

```
788     private void btnConectarActionPerformed(java.awt.event.ActionEvent evt) {
789         confBanco.setDriver(txtDriverBanco.getText());
790         confBanco.setUrl(txtUrl.getText().toString());
791         confBanco.setUrlCompleto(urlBanco + txtUrl.getText().toString());
792         confBanco.setUsuario(txtUsuario.getText().toString());
793         confBanco.setSenha(txtSenha.getText().toString());
794         carregarComboDatabases();
795     }
```

Ao realizar a chamada do trecho Código 5 o framework vai fazer a conexão com SGBD e realizar a chamada do método *carregarComboDatabases()* responsável por carregar na interface com o usuário uma caixa de combinação contendo os nomes dos bancos de dados disponíveis no SGBD selecionado. A lista dos bancos de dados disponíveis no banco da dados é armazenada no objeto *listaDatabases* ilustrada no Código 6. Neste trecho de código é utilizado dos conceitos de reflexão para obter os nomes dos bancos de dados disponíveis. Para tal ocorre a chamada do método *getListaDatabase* da classe *bancoService*. A classe *bancoService* é responsável pelos métodos que fazem as consultas em tempo de execução de acordo com o SGBD selecionado. Na instanciação da classe passa-se os argumentos uma classe e um objeto da classe *ConfiguracaoBE*.

Código 6. Geração da Lista de Bancos de Dados Disponíveis.

```
984     @SuppressWarnings("static-access")
985     private void carregarComboDatabases() {
986         bancoService = new BancoService(confBanco, bancoConfig);
987         List<String> listaDatabases = bancoService.getListaDatabases();
988         for (int i = 0; i < listaDatabases.size(); i++) {
989             cmbDatabase.addItem(listaDatabases.get(i));
990         }
991         cmbDatabase.setSelectedIndex(-1);
992     }
```

O trecho Código 7, abaixo, mostra como é montada a lista de bancos de dados disponíveis. Na classe *BancoDAO* são montadas as instruções SQL para selecionar os metadados do SGBD.

Código 7. Geração da Lista de Banco de Dados.

```
96      public List<String> getListaDatabases() {
97          try {
98              bancoDAO = new BancoDAO(configuracaoBE, conexao);
99              ResultSet rs = bancoDAO.listarDatabases();
100             List<String> listaDatabases = new ArrayList<>();
101             while (rs.next()) {
102                 listaDatabases.add(
103                         rs.getString(
104                 ReflectionUtils.getValorCampo(bancoConfiguracao, "CAMPO_DATABASE", new Object()).toString()));
105             }
106             return listaDatabases;
107         } catch (SQLException ex) {
108             throw new SQLExcecao(ex);
109         }
110     }
111
```

O método *listarDatabases* apresentado no Código 8 monta uma SQL de acordo com SGBD informado a partir de uma classe de interface (Código 9), que já tem valores predefinidos.

Código 8. Método *listarDatabases*.

```
44      public ResultSet listarDatabases() {
45          try {
46              carregarClasse();
47              sql = ReflectionUtils.getValorCampo(bancoConfig, "DATABASES", new Object()).toString();
48              preparedStatement = conexao.prepareStatement(sql);
49              return preparedStatement.executeQuery();
50          } catch (SQLException ex) {
51              throw new SQLExcecao(ex);
52          } catch (IllegalArgumentException ex) {
53              throw new ArgumentoIlegalExcecao(ex);
54          } catch (SecurityException ex) {
55              throw new SegurancaExcecao(ex);
56          }
57      }
```

No Código 9 é apresentado uma *interface* para o SGBD MySQL. Nesta *interface* se encontra as instruções SQL de metadados para obter os bancos de dados, as tabelas e os seus respectivos campos. Tal *interface* permite selecionar uma determinada tabela e verificar se existe chave estrangeira. Também são oferecidos métodos para verificar alguns atributos do banco, por exemplo: driver do banco, dialeto

do Hibernate, url, campos das tabelas de metadados. Tais estruturas são utilizadas para se definir um sistema reflexivo em tempo de execução e desta forma deixar os códigos mais dinâmicos.

Código 9. Interface da Classe.

```
3  public interface MySQLConstantes {
4   public static final String DRIVER = "com.mysql.jdbc.Driver";
5   public static final String DIALECT_HIBERNATE = "org.hibernate.dialect.MySQLDialect";
6   public static final String URL = "jdbc:mysql://";
7   public static final String DATABASES = "select * from `information_schema`.`SCHEMATA`";
8   public static final String TABELAS = "select * from `information_schema`.`TABLES` where `TABLE_SCHEMA` = ?";
9   public static final String SELECIONAR_TABELA = "select * from `information_schema`.`TABLES` where `TABLE_SCHEMA`
10  public static final String CAMPOS = "select * from `information_schema`.`COLUMNS` where `TABLE_SCHEMA` = ? and `
11  public static final String CAMPO_CHAVE_ESTRANGEIRAS = "select * from information_schema.`KEY_COLUMN_USAGE` where
12  public static final String CAMPO_DATABASE = "SCHEMA_NAME";
13  public static final String CAMPO_TABELA = "TABLE_NAME";
14  public static final String CAMPO_NOME = "COLUMN_NAME";
15  public static final String CAMPO_NULLABLE = "IS_NULLABLE";
16  public static final String CAMPO_TIPO = "DATA_TYPE";
17  public static final String CAMPO_TAMANHO = "CHARACTER_MAXIMUM_LENGTH";
18  public static final String CAMPO_NUMERIC = "NUMERIC_PRECISION";
19  public static final String CAMPO_PRECISAO = "NUMERIC_SCALE";
20  public static final String CAMPO_KEY = "COLUMN_KEY";
21  public static final String CAMPO_EXTRA = "EXTRA";
22  public static final String CAMPO_REFERENCIA_TABELA = "REFERENCED_TABLE_NAME";
23  public static final String CAMPO_REFERENCIA_COLUNA = "REFERENCED_COLUMN_NAME";
24 }
25
```

O usuário poderá selecionar qual esquema (*schema*) ou banco de dados (*database*) que deseja trabalhar e selecionar a opção *Salvar*. Tal opção permite salvar as configurações definidas. O sistema gera um arquivo *config.ini*. Este arquivo permite ao usuário utilizar as mesmas configurações em outro momento.

Após o usuário escolher qual banco deseja utilizar, o framework realizará uma nova conexão para listar todas as tabelas e os seus atributos. Neste momento também verificará as chaves primária e as chaves estrangeiras.

Esse processo em bem extenso e utiliza muita reflexão computacional. Primeiramente são listadas todas as tabelas do banco de dados. Depois são listados todos os campos de cada tabela e realizada uma verificação dos tipos de cada campo e os seus respectivos nomes. No Código 10 é apresentado o método *getListaTabela* responsável por listar todas as tabelas definidas em um determinado banco de dados. Para realizar a listagem das tabelas foram definidas duas classes contendo a descrição dos atributos necessários para listar uma tabela (*TabelaBE*) e os atributos dos respectivos campos da tabela (*AtributoBE*).

Código 10. Método *getListaTabela*.

```
33  public List<TabelaBE> getListaTabela() {
34      try {
35          bancoDAO = new BancoDAO(configuracaoBE, conexao);
36          ResultSet rsTabelas = bancoDAO.listarTabelas(configuracaoBE.getDatabase());
37          TabelaBE tabelaBE;
38          List<TabelaBE> tabelaBEs = new ArrayList<>();
39          while (rsTabelas.next()) {
40              tabelaBE = new TabelaBE();
41              tabelaBE = carregarTabela(rsTabelas);
42              List<AtributosBE> listaAtributos = new ArrayList<>();
43              ResultSet rsAtributos = bancoDAO.listarCamposTabelas(configuracaoBE.getDatabase(), tabelaBE.getTabe
44              AtributosBE atributosBE;
45              while (rsAtributos.next()) {
46                  atributosBE = new AtributosBE();
47                  atributosBE.setNameTable(rsAtributos.getString(ReflectionUtils.getValorCampo(bancoConfiguracao,
48                  atributosBE.setName(Funcoes.gerarNomeAtributo(rsAtributos.getString(ReflectionUtils.getValorCam
49                  atributosBE.setColunaKey(rsAtributos.getString(ReflectionUtils.getValorCampo(bancoConfiguracao,
50                  atributosBE.setDataType(rsAtributos.getString(ReflectionUtils.getValorCampo(bancoConfiguracao,
51                  if (rsAtributos.getInt(ReflectionUtils.getValorCampo(bancoConfiguracao, "CAMPO_TAMANHO", new Ob
52                      atributosBE.setLenght(rsAtributos.getInt(ReflectionUtils.getValorCampo(bancoConfiguracao, "
53                      atributosBE.setPrecision(rsAtributos.getInt(ReflectionUtils.getValorCampo(bancoConfiguracac
54                  } else {
55                      atributosBE.setLenght(rsAtributos.getInt(ReflectionUtils.getValorCampo(bancoConfiguracao, "
```

A classe TabelaBE consiste na definição dos atributos necessários para armazenar as informações das tabelas pertencentes a um determinado banco de dados. Tal classe define o nome da tabela, a classe que contém os atributos de modelo necessários ao usuário, o nome do formulário que será gerado e atributos lógicos para informar se será ou não gerado códigos da referida tabela para Cadastro (*menuCadastro*), Consulta (*menuConsulta*) e Relatório (*menuRelatório*).

Código 11. Classe *TabelaBE*.

```
2
3   import java.util.List;
4
5   public class TabelaBE {
6
7       private String tabela;
8       private String classe;
9       private String formulario;
10      private boolean menuCadastro;
11      private boolean menuConsulta;
12      private boolean menuRelatorio;
13      private List<AtributosBE> listaAtributos;
14
```

A classe *AtributoBE*, ilustrada no tem por objetivo definir os atributos que descrevem as características de cada um dos campos que compõem cada uma das tabelas do banco de dados. Com base em tais atributos, posteriormente, serão definidas as características das caixas textos e dos valores que serão manipulados no sistema.

Código 12. Classe *AtributoBE*.

```
5   public class AtributosBE {
6
7       private String name;
8       private String nameTable;
9       private String valores;
10      private boolean filtro = false;
11      private boolean requerido = false;
12      private boolean editavel = true;
13      private String dataType;
14      private String colunaKey;
15      private String tipoCampo = "Campo texto";
16      private String tipoFormatacao = "";
17      private int scale = 0;
18      private int precision = 0;
19      private int lenght = 255;
20      private String tabela;
21      private boolean inserttable = true;
22      private boolean updatable = true;
23      private boolean unique = false;
24      private boolean notnull = true;
25      private boolean chaveEstrangeira;
26      private ChaveEstrangeiraBE chaveEstrangeiraBE;
27      private boolean primaryKey = false;
28      private TipoChavePrimaria primaryGenerator = TipoChavePrimaria.INCREMENT;
29      private String columnDefinition = "";
```

Após terem sido preenchidas as definições do sistema, o Usuário poderá mudar para a aba *Selecione as tabelas,* conforme ilustrado na Figura 4. Nesta aba encontram-se as ações responsáveis por mostrar todas as tabelas, por exemplo, o nome da tabela no banco, o nome da classe referente á tabela, nome do formulário de movimentação, atributos para selecionar em qual o menu que será apresentado a opção de escolha para o usuário. Nesta interface de usuário, o mesmo não poderá alterar os atributos como o nome da tabela e nome da classe relacionada, por motivos de padronização de código fonte.

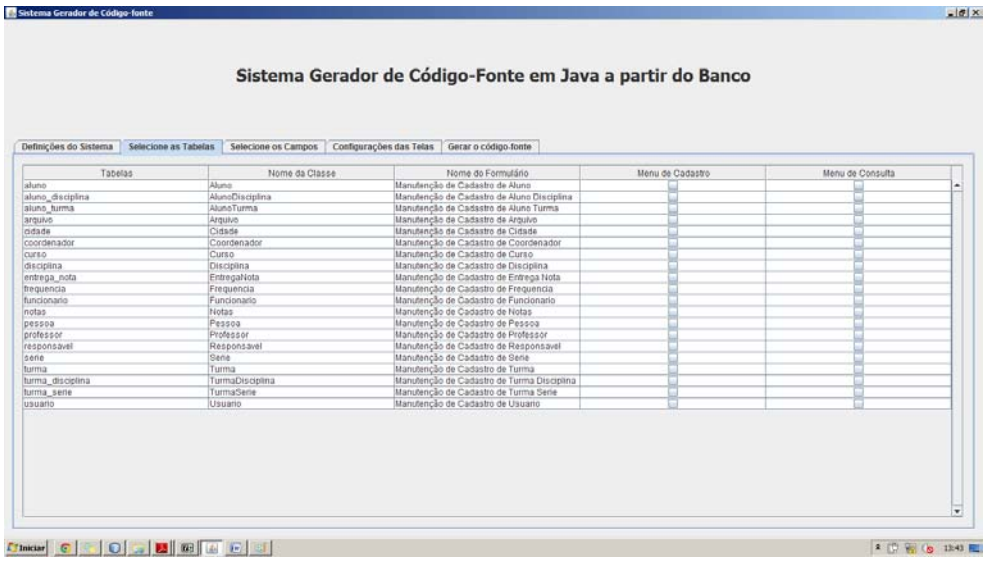


Figura 4. Aba Seleciona Tabelas

Na aba *Selecione os Campos* (Figura 5) o usuário poderá visualizar todos os campos de cada tabela e verificar qual campo é chave estrangeira, ou se o campo é chave primária e/ou único. O usuário poderá alterar o rótulo para o campo no formulário, qual o tipo de campo, se é campo formatado, se é requerido, entre outros tipos. Opções para informar se o campo é filtro de pesquisa e/ou evitável no formulário também são fornecidas.

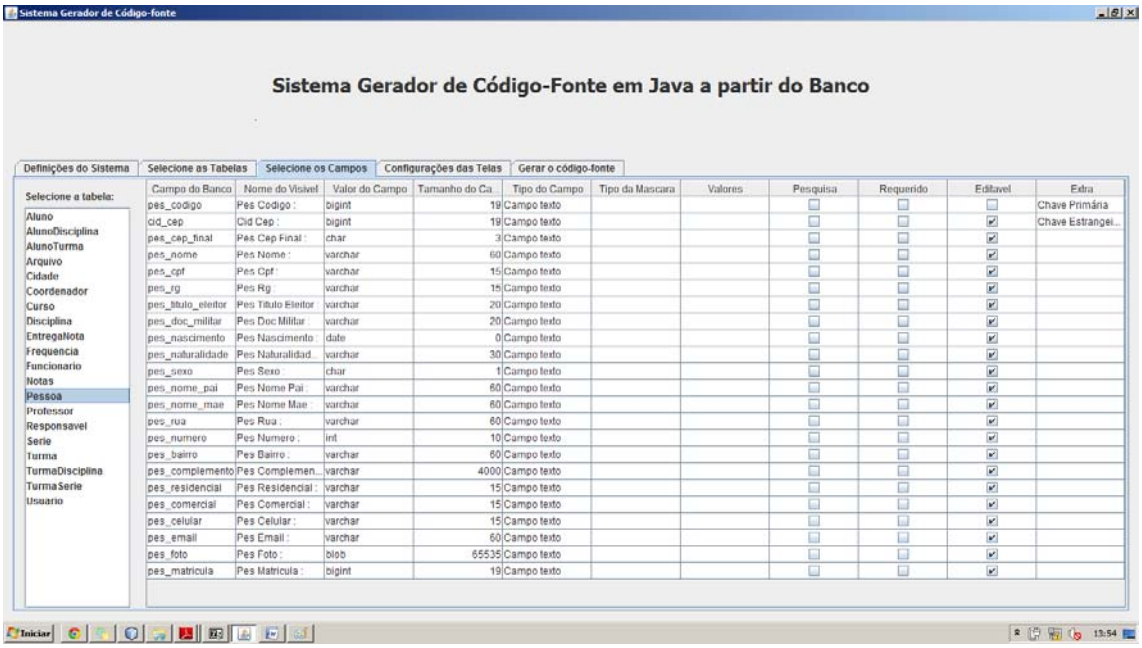


Figura 5. Aba Selecione os Campos.

Na aba de *Gerar o código-fonte*, o usuário irá escolher o local onde vai ser gerado os códigos fontes e poderá também definir o pacote padrão da aplicação. A Figura 6 ilustra a interface gráfica apresentada ao usuário que define as configurações da etapa de geração de código-fonte.

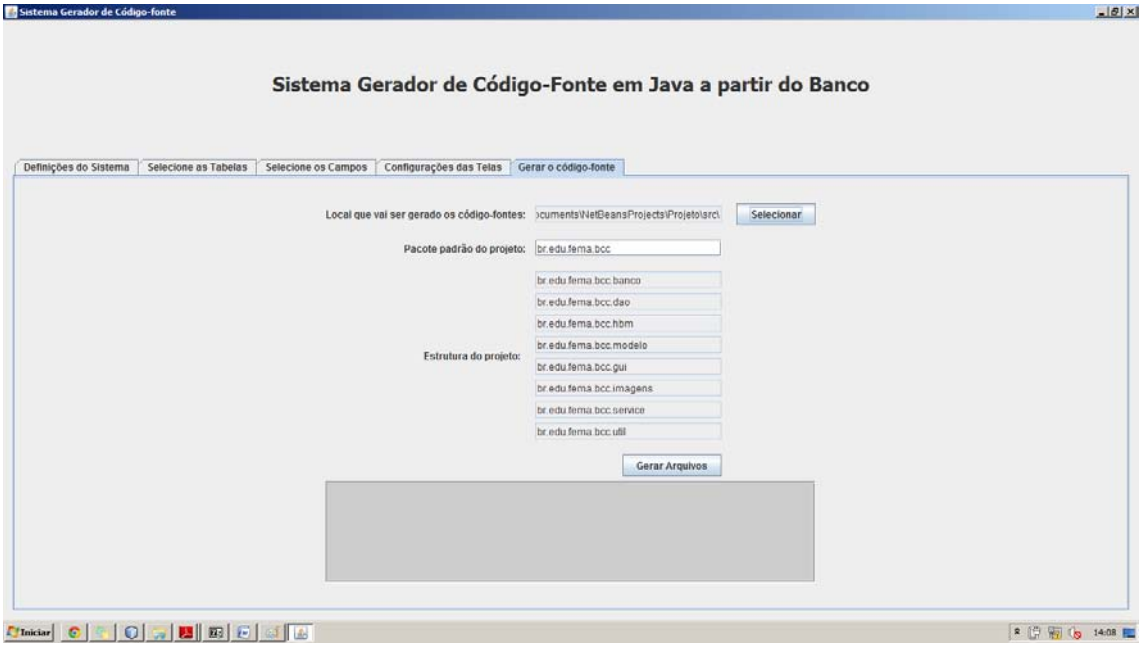


Figura 6. Interface gráfica etapa Gerar Código-Fonte

Após o usuário selecionar o local e o pacote padrão. O *framework* monta uma estrutura de pastas:

- **Banco**: local onde vai se encontrar o arquivo de conexão do Hibernate;
- **DAO**: pasta utilizada para armazenar os arquivos referentes as classes de comunicação com o banco de dados, responsável pelos métodos *salvar, editar*, *selecionar*, *listar* e *excluir*;
- **HBM**: pasta que conterá os arquivos no formato XML responsáveis pelo mapeamentos do Hibernate;
- **MODELO**: pasta que conterá os arquivos JAVA que descrevem as classe de Modelo, responsáveis pelo mapeamento objeto/relacional do banco de dados;
- **GUI**: pasta que contém as interfaces gráficas, para a parte visual, os formulários que constituem o sistema;
- **IMAGENS**: pasta para armazenar as imagens utilizadas no sistema
- **UTIL**: pasta responsável por guardar utilitários de validações e conversões de dados.

Depois selecionada a opção *Gerar Arquivos* inicia-se a geração dos códigos-fontes. O framework começa a gerar os arquivos a partir da chamada do construtor do *GerarArquivoJava* passando três argumentos que são o *ConfiguracaoBE*, lista de *TabelaBE* e *ArquivosBE*.

Código 13. Classe *GerarArquivoJava*.

```
15  public class GerarArquivosJava {
16
17      public GerarArquivosJava(ConfiguracaoBE configuracaoBE, List<TabelaBE> listaTabelas, ArquivosBE arquivosBE) {
18          gerarMapeamento = new GerarHbmHibernateJava(arquivosBE);
19          String[] mapeamento = gerarMapeamento.gerarMapeamento(listaTabelas);
20          String[] tabelas = getTabelas(listaTabelas);
21          gerarRegengHibernate = new GerarRegengHibernateJava(arquivosBE, tabelas, configuracaoBE.getDatabase());
22          gerarRegengHibernate.gerarRegengHibernate();
23          gerarHibernate = new GerarCfgHibernateJava(configuracaoBE, arquivosBE, mapeamento);
24          gerarHibernate.gerarConfiguracaoHibernate();
25          gerarModelos = new GerarModelos(arquivosBE);
26          gerarModelos.gerarModelos(listaTabelas);
27          gerarDAOJava = new GerarDAOJava(arquivosBE);
28          gerarDAOJava.gerarDAOS(listaTabelas);
29          gerarUtilJava = new GerarUtilJava(arquivosBE);
30          gerarUtilJava.gerarUtils();
31      }
```

Depois de instanciada a classe inicia-se o processo de geração. Inicialmente é realizada a chamada de uma classe referente a geração dos mapeamentos do XML do Hibernate. Nesse processo é feito todo o mapeamento do banco conforme é realizada a leitura dos metadados do banco. Para tal foi realizado uma padronização do código XML de mapeamento do Hibernate. A Figura 7 ilustra o código XML padronizado.

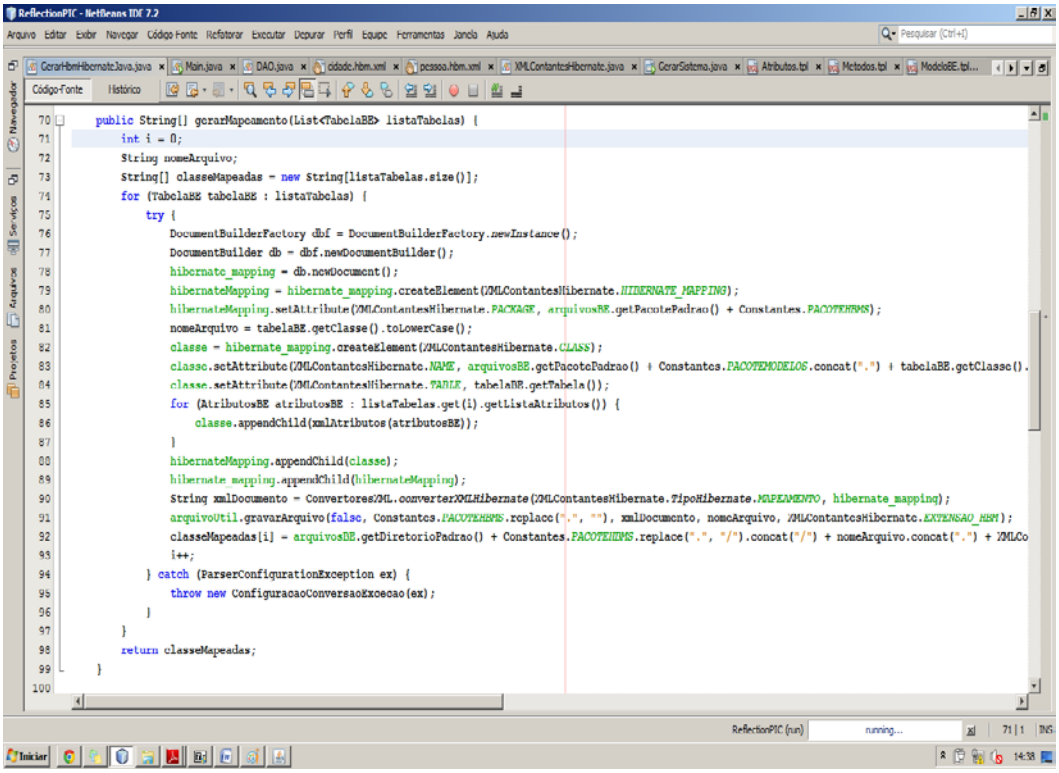
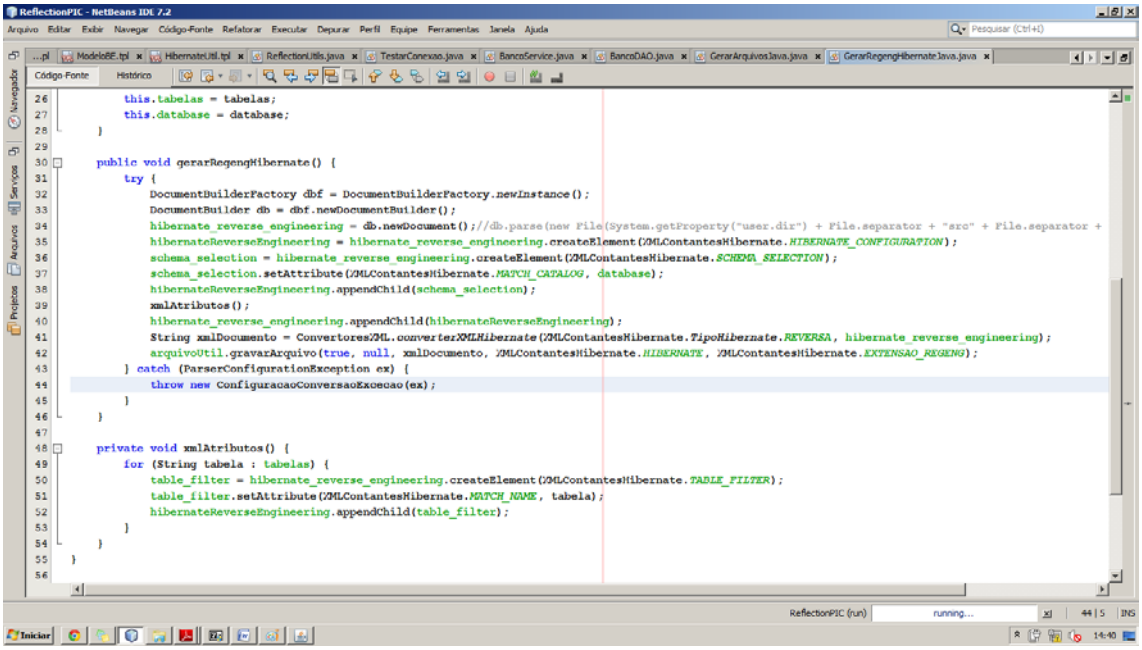


Figura 7. Padronização do XML.

Depois de realizado o mapeamento do XML, gera-se o arquivo de reversão do Hibernate e, na sequencia, o arquivo de configuração do Hibernate. O método utilizado para geração da engenharia reversa do Hibernate é descrito na Figura 8.



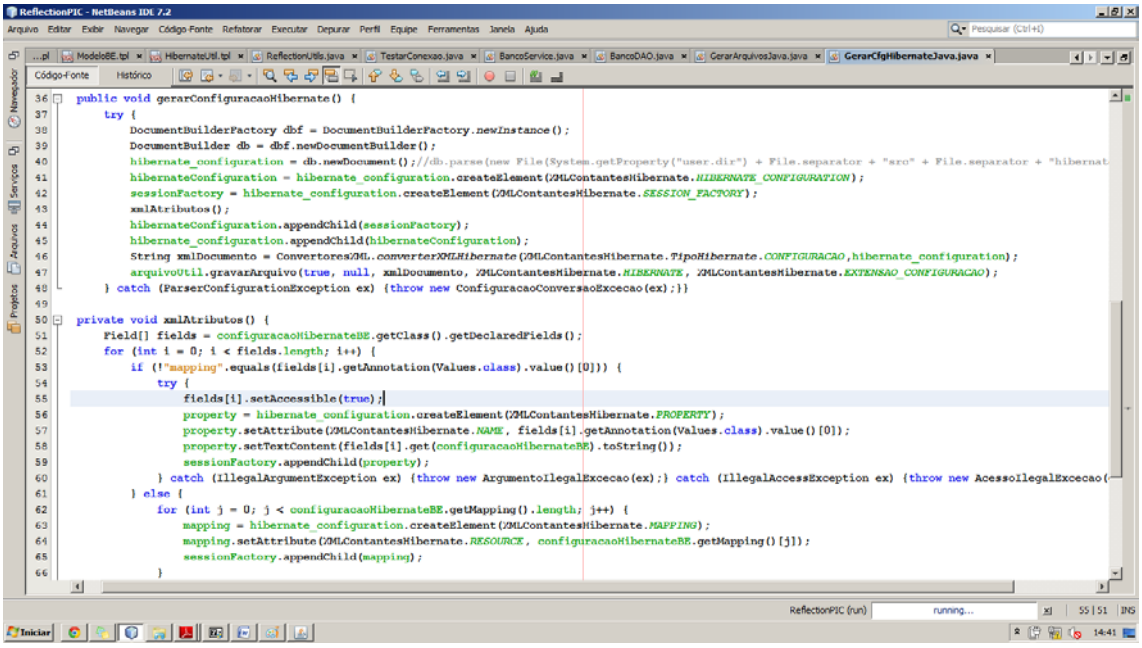
```
        this.tabelas = tabelas;
        this.database = database;
    }

    public void gerarRegengHibernate() {
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            hibernate_reverse_engineering = db.newDocument();//db.parse(new File(System.getProperty("user.dir") + File.separator + "src" + File.separator +
            hibernateReverseEngineering = hibernate_reverse_engineering.createElement(XMLContantesHibernate.HIBERNATE_CONFIGURATION);
            schema_selection = hibernate_reverse_engineering.createElement(XMLContantesHibernate.SCHEMA_SELECTION);
            schema_selection.setAttribute(XMLContantesHibernate.MATCH_CATALOG, database);
            hibernateReverseEngineering.appendChild(schema_selection);
            xmlAtributos();
            hibernate_reverse_engineering.appendChild(hibernateReverseEngineering);
            String xmlDocumento = ConvertoresXML.converterXMLHibernate(XMLContantesHibernate.TipoHibernate.REVERSA, hibernate_reverse_engineering);
            arquivoUtil.gravarArquivo(true, null, xmlDocumento, XMLContantesHibernate.HIBERNATE, XMLContantesHibernate.EXTENSAO_REGENG);
        } catch (ParserConfigurationException ex) {
            throw new ConfiguracaoConversaoExcecao(ex);
        }
    }

    private void xmlAtributos() {
        for (String tabela : tabelas) {
            table_filter = hibernate_reverse_engineering.createElement(XMLContantesHibernate.TABLE_FILTER);
            table_filter.setAttribute(XMLContantesHibernate.MATCH_NAME, tabela);
            hibernateReverseEngineering.appendChild(table_filter);
        }
    }
}
```

Figura 8. Método *gerarRegengHibernate*().

Já na Figura 9 são descritos os métodos *geraConfiguracaoHibernate* e *xmlAtributos* responsáveis respectivamente pela geração da configuração do Hibernate e manipulação dos atributos das interfaces gráficas que serão geradas posteriormente.



```
public void gerarConfiguracaoHibernate() {
    try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        hibernate_configuration = db.newDocument();//db.parse(new File(System.getProperty("user.dir") + File.separator + "src" + File.separator + "hibernat
        hibernateConfiguration = hibernate_configuration.createElement(XMLContantesHibernate.HIBERNATE_CONFIGURATION);
        sessionFactory = hibernate_configuration.createElement(XMLContantesHibernate.SESSION_FACTORY);
        xmlAtributos();
        hibernateConfiguration.appendChild(sessionFactory);
        hibernate_configuration.appendChild(hibernateConfiguration);
        String xmlDocumento = ConvertoresXML.converterXMLHibernate(XMLContantesHibernate.TipoHibernate.CONFIGURACAO,hibernate_configuration);
        arquivoUtil.gravarArquivo(true, null, xmlDocumento, XMLContantesHibernate.HIBERNATE, XMLContantesHibernate.EXTENSAO_CONFIGURACAO);
    } catch (ParserConfigurationException ex) {throw new ConfiguracaoConversaoExcecao(ex);}}

private void xmlAtributos() {
    Field[] fields = configuracaoHibernateBE.getClass().getDeclaredFields();
    for (int i = 0; i < fields.length; i++) {
        if (!"mapping".equals(fields[i].getAnnotation(Values.class).value()[0])) {
            try {
                fields[i].setAccessible(true);
                property = hibernate_configuration.createElement(XMLContantesHibernate.PROPERTY);
                property.setAttribute(XMLContantesHibernate.NAME, fields[i].getAnnotation(Values.class).value()[0]);
                property.setTextContent(fields[i].get(configuracaoHibernateBE).toString());
                sessionFactory.appendChild(property);
            } catch (IllegalArgumentException ex) {throw new ArgumentoIllegalExcecao(ex);} catch (IllegalAccessException ex) {throw new AcessoIllegalExcecao(
        } else {
            for (int j = 0; j < configuracaoHibernateBE.getMapping().length; j++) {
                mapping = hibernate_configuration.createElement(XMLContantesHibernate.MAPPING);
                mapping.setAttribute(XMLContantesHibernate.RESOURCE, configuracaoHibernateBE.getMapping()[j]);
                sessionFactory.appendChild(mapping);
            }
```

Figura 9. Métodos *gerarConfiguraçãoHibernate* e *xmlAtributos*.

Depois de configurado o Hibernate, o *framework* vai gerar as classes de Modelo do banco. Nesta etapa o *framework* utiliza alguns arquivos de estrutura já padronizados para facilitar a geração das classes (Figura 10).

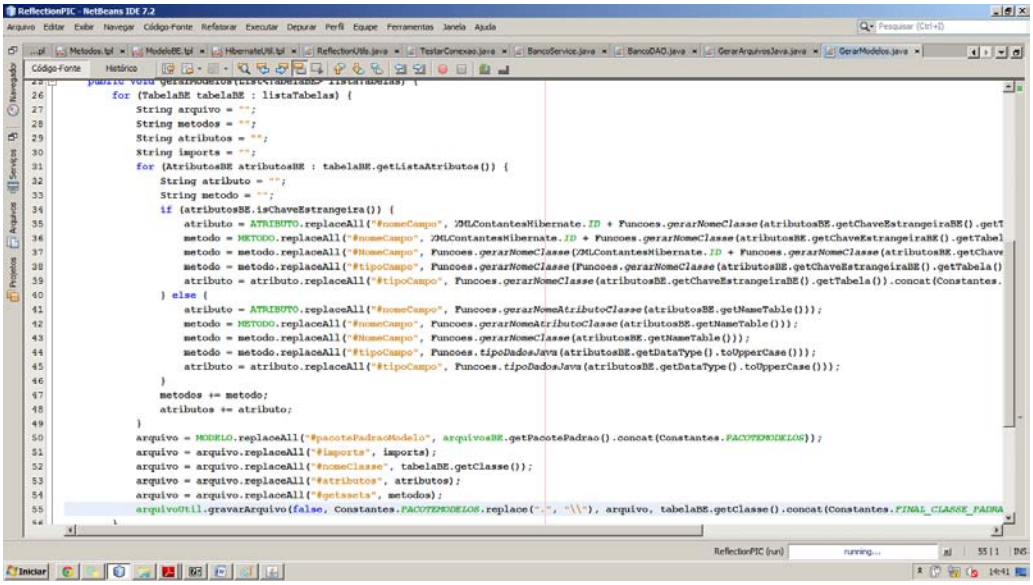


Figura 10. Trecho de código geração de Classes Modelos.

Por fim, gera-se as classes DAO que seguem mesmo padronização dos arquivos. E também são gerados os filtros de pesquisas conforme o usuário selecionar.

4. Conclusão

Neste trabalho foram apresentados conceitos de reflexão computacional e um estudo de caso fundamentado na construção de um *framework* para geração automática de aplicações comerciais foi realizado. Com base no estudo de caso desenvolvido foi possível destacar a vantagem de utilizar reflexão computacional. No Capitulo 2 foram mostrados o conceito de reflexão computacional, seus modelos e conceitos de orientação a objetos, que é a base para o estudo de reflexão, também apresentou-se as propriedades da classe Reflection definida para a linguagem JAVA.

No Capítulo 3 foi apresentada a arquitetura do *framework* para geração de aplicações comerciais concebido, a construção do mesmo e os conceitos de reflexão computacional empregados.

No decorrer deste trabalho foi possível avaliar a reflexão, pois possibilita uma forma dinâmica de programar. Com o uso de tal técnica ocorre uma redução na programação uma vez que o código fica dinâmico e pode adaptar-se a diferentes situações, em tempo de execução.

Com base na pesquisa desenvolvida, várias vertentes para futuros trabalhos podem ser identificadas, Como possíveis trabalhos futuros, pode-se apontar:

- Melhorias no framework para a definição das interfaces gráficas geradas;
- Desenvolvimento de um módulo para auxiliar a geração de código para realizar movimentações e atualizações de tabelas relacionadas;
- Geração automática de relatórios;
- Pesquisa para aplicação de reflexão em regra de negócios, como interesse sistêmico;

Referências Bibliográficas

Arquitetura Reflexiva. **Conceitos de arquitetura reflexiva**. Disponível em: http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem991/T1/alex/2/arquitetura/arquitetura-reflexiva.htm. Acessado em 11 de Jun. 2012

BARTH, F.J.. **Utilização da Reflexão Computacional para implementação de aspectos não funcionais em um gerenciador de arquivos distribuídos.** Trabalho de Conclusão de Curso, Universidade Regional de Blumenau. 2000

CAMOLESI, A.R.; NETO, J.J. **Modelagem Adaptativa de Aplicções Complexas**. XXX Conferencia Latinoamericana de Informática - CLEI'04. Arequipa - Peru, Setiembre 27 - Octubre 1, 2004.

CASACHI, R. A. **Aplicação do Paradigma de Programação Orientada a Aspectos no Desenvolvimento de Software.** Trabalho de Conclusão de Curso, Bacharelado em Ciência da Computação, FEMA-IMESA, 2011.

Conceitos Básicos. Disponível em: http://www2.dbd.puc rio.br/pergamum /tesesabertas/0115636_02_cap_02.pdf. Acessado em 07 de Junho. 2012

FERREIRA, A.B. H. **Minidicionário da Língua portuguesa**. Editora Nova Fronteira, 4 Ed, 2000

JENKOV, j. **Java Reflection Tutorial.** Disponível em: http://tutorials.jenkov.com /java-reflection/index.html.

KICZALES, G. ; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, V.; LOINGTIER, JM. **Aspect-Oriented Programming**. In: European Conference on Object-Oriented Programming (ECOOP), 06,1997. Finlândia. Anais Springer-Verlag LNCS 1241, 06, 1997.

Lecture Notes in Computer Science. **Meta-Level Architectures and Reflection.** Springer Verlag, 1999.

Lecture Notes in Computer Science. **Reflection and Software Engineering.** Springer Verlag, 2000.

MAES, P. **Concepts and experiments in computational reflection.** ACM Sigplan Notices, v. 22, n. 12, p. 147-155, Dec. 1987

NETO, J.J. **Contribuições à metodologia de construção de compiladores**. Tese de Livre Docência, USP, São Paulo, 1993.

Oracle. **Trail: The Reflection API**. Uses of Reflection. Disponível em: docs.oracle.com/javase/tutorial/reflect/index.html. Acessado em 12 de Jun. 2012

PISTORI, H. **Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações**. Tese de Doutorado, USP, São Paulo, 2003.

RICARTE, I. L. M. **Programação Orientada a Objetos: Uma Abordagem com Java**, Universidade Estadual de Campinas (UNICAMP). 2001

SALLEM, M. A. S. **Adapta: um arcabouço para o desenvolvimento de aplicações distribuídas adaptativas.** Trabalho de Pós-Graduação, Universidade Federal do Maranhão. 2007

SILVA, R. C. **RStabilis: Uma Máquina Reflexiva de Busca**, Dissertação de mestrado, Universidade Estadual de Campinas (UNICAMP). 1997

SMITH, B. C. **Reflection and Semanticsin a Procedural Language**. In Morgan Kaufmann, editor, The Knowledge Representation Enterprise, pp. 31-39. R.J. Brachman and H.S. Levisque, 1985.

SOUSA, F. C. **Utilização da Reflexão Computacional para implementação de um monitor de software orientado a objetos em Java**. Trabalho de Conclusão de Curso, Universidade Regional de Blumenau. 2002

Tutoriais Admin. **Tutorial Java: O que é Java**. JavaFree. Disponível em: http://javafree.uol.com.br/artigo/871498/Tutorial-Java-O-que-e-Java.html. Acessado em 17 de abr. 2012

YOURDON, E. **Análise Estruturada Moderna**. Editora Campus, 1990.

Jefferson Simão Gonçalves	**Dr. Almir Rogério Camolesi**
Orientado	**Orientador**