UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

GERADOR DE CÓDIGO JSP BASEADO EM PROJETO DE BANCO DE DADOS

MAICON KLUG

BLUMENAU
2007

2007/1-26

MAICON KLUG

GERADOR DE CÓDIGO JSP BASEADO EM PROJETO DE BANCO DE DADOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof[a]. Joyce Martins, Mestre - Orientadora

BLUMENAU
2007

2007/1-26

GERADOR DE CÓDIGO JSP BASEADO EM PROJETO DE BANCO DE DADOS

Por

MAICON KLUG

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof[a]. Joyce Martins, Mestre – Orientadora, FURB
	_____
Membro:	Prof. Adilson Vahldick, Especialista – FURB
	_____
Membro:	Prof. Alexander Roberto Valdameri, Mestre – FURB

Blumenau, 03 de julho de 2007

Dedico este trabalho a minha família, pai, mãe e irmão, que sempre estiveram comigo durante todos esses anos de faculdade, a minha namorada, pela compreensão e por ter sido meu ombro amigo nos momentos mais difíceis, aos meus amigos, pelo auxílio e incentivo nas aulas e nos trabalhos em grupo, e especialmente a minha orientadora, pelos conselhos e apoio na realização deste trabalho.

AGRADECIMENTOS

À minha família, que sempre esteve comigo em todos os momentos, compartilhando das horas alegres e tristes.

À minha namorada, Mônica, que esteve sempre me apoiando e incentivando, fazendo com que eu enfrentasse os desafios e não desistisse de forma nenhuma.

Aos meus amigos, André, Gilson e Leonardo, pela ajuda e incentivo no desenvolvimento desse trabalho, assim como nos demais trabalhos das disciplinas do curso.

À minha orientadora, Joyce Martins, pelas reuniões, pelo apoio e principalmente por ter acreditado em mim durante todo o desenvolvimento deste trabalho.

Sonho que se sonha só é só um sonho que se sonha só, mas sonho que se sonha junto é realidade.

Prelúdio, Raul Seixas

RESUMO

O presente trabalho apresenta uma solução desenvolvida em forma de ferramenta, para efetuar a geração de páginas JSP no padrão MVC, a partir do metadados de um banco de dados relacional. Através da API JDBC é extraído o dicionário de dados dos SGBDs Oracle, Microsoft SQL Server e MySQL. Utilizando as informações extraídas do metadados, a ferramenta permite ao desenvolvedor criar um projeto, configurar e salvar as definições de formulários, grupos e relatórios. As páginas JSP a serem geradas, utilizando as informações configuradas, são definidas através de *templates* e a geração é realizada utilizando o motor de *templates* Velocity. As páginas geradas possuem funcionalidades de inclusão, alteração, exclusão e consulta. A ferramenta é aplicada em um estudo de caso que descreve um sistema de locação de filmes.

Palavras-chave: Geradores de código. *Templates*. Banco de dados. JSP. MVC.

ABSTRACT

This monography presents a solution developed as a tool, to generate JSP pages using the design pattern MVC, from the metadata of a relational database. Through JDBC API, the data dictionary is extracted from the DBMSs Oracle, Microsoft SQL Server and MySQL. Using the extracted information from the metadata, the tool allows the developer to create a project, to configure and to save the form definitions, groups and reports. JSP pages to be generated, using the configured information, are defined through templates and the generation is made by using the engine of templates Velocity. The generated pages have these functionalities: insert, update, remove and query. The tool is applied in a case study that describes a system for a movie rental store.

Key-words: Code generators. Templates. Database. JSP. MVC.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

ASP – *Active Server Pages*

AST – *Abstract Syntax Tree*

CASE – *Computer-Aided Software Engineering*

CSS – *Cascading Style Sheets*

DAO – *Data Access Object*

EJB – *Enterprise JavaBeans*

GUI – *Graphical User Interface*

HTML – *HyperText Markup Language*

HTTP – *HyperText Transfer Protocol*

J2EE – *Java 2 Enterprise Edition*

JDBC – *Java Data Base Connectivity*

JDK – *Java Development Kit*

JNI – *Java Native Interface*

JSP – *JavaServer Pages*

JVM – *Java Virtual Machine*

MVC – *Model-View-Controller*

ODBC – *Open Database Connectivity*

PHP – *Php Hypertext Preprocessor*

SGBD – *Sistema Gerenciador de Banco de Dados*

SQL – *Structured Query Language*

SWT – *Standard Widget Toolkit*

UML – *Unified Modeling Language*

URL – *Uniform Resource Locator*

VTL – *Velocity Template Language*

XHTML – *eXtensible HyperText Markup Language*

XML – *eXtensible Markup Language*

XSL – *eXtensible Stylesheet Language*

SUMÁRIO

1 INTRODUÇÃO

O acesso à Internet cresce com o passar dos anos devido às vantagens que ela proporciona aos usuários. A busca por notícias e documentos, a publicação de dados eletrônicos ou até mesmo o acesso a sistemas de gestão, são algumas das atividades que podem ser realizadas na rede. Todas essas atividades são processadas de forma transparente ao usuário, pois a única informação que ele tem é o endereço eletrônico da página que deseja acessar. Para Abiteboul, Buneman e Suciu (2000, p. 1), a maioria das pessoas vê esses dados como simples documentos web, sem conhecer como eles são realmente construídos. Esses arquivos eletrônicos estão cada vez mais sendo gerados automaticamente a partir de bancos de dados, e essa geração automática permite publicar em páginas web grandes volumes de dados.

Muitos dos softwares existentes no mercado utilizam-se de bancos de dados para o armazenamento das informações, e nas aplicações web não ocorre de modo diferente. Por mais que seja transparente ao usuário, são as informações que ele insere nas telas ou interfaces dos sistemas que vão alimentar a base de dados, e são as suas ações que vão manipular essas informações. Assim, o processo de interação do usuário com a interface do sistema precisa ser projetado e as possíveis exceções precisam ser previstas e tratadas, a fim de transmitir ao usuário segurança na utilização do aplicativo. Mas esse desenvolvimento demanda tempo e esforço, gerando um trabalho muito repetitivo para o desenvolvedor. Para aumentar a produtividade, a qualidade e a padronização do código fonte a ser desenvolvido, podem ser utilizadas ferramentas para a geração automática de código. Dessa forma, funções mais simples, como a criação de interfaces e o acesso a banco de dados, podem ser criadas a partir de um gerador de código, deixando o tratamento das rotinas específicas de negócio para o desenvolvedor.

Diante do exposto, o presente trabalho apresenta o desenvolvimento de uma ferramenta para a geração de páginas JSP a partir da definição de uma base de dados em um banco de dados relacional. Com o auxílio da API JDBC, a ferramenta realiza a leitura do metadados[1] de uma base de dados dos SGBDs relacionais Oracle, Microsoft SQL Server e MySQL. Através do motor de *templates*[2] Velocity, as páginas JSP são geradas com

[1] Metadados são dados que descrevem outros dados. Em bancos de dados relacionais, são estruturas que definem as tabelas para o armazenamento das informações. De forma análoga, são tabelas que armazenam as definições de outras tabelas (RUMBAUGH et al., 1994, p. 94). Metadados, dicionário de dados e estrutura da base de dados são utilizados como sinônimos no decorrer deste trabalho.

[2] *Templates* são modelos que servem de base para geração do código de saída.

funcionalidades de inserção, alteração, exclusão e consulta, seguindo o padrão estabelecido pela arquitetura MVC. Através do uso de *templates*, o código gerado não fica incluído no código fonte da ferramenta, possibilitando uma maior flexibilidade ao desenvolvedor, que poderá optar em gerar o código com diferentes *layouts* de tela e até mesmo em diferentes linguagens.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo desenvolver uma ferramenta para a automatização do processo de desenvolvimento de software, gerando páginas JSP a partir do dicionário de dados de um SGBD.

Os objetivos específicos do trabalho são:

a) efetuar a leitura do dicionário de dados dos SGBDs Oracle, Microsoft SQL Server e MySQL, através da API JDBC;
b) utilizar *templates* na definição do formato do código de saída;
c) gerar páginas JSP com funcionalidades de inclusão, alteração, exclusão e consulta;
d) gerar o código de saída seguindo o padrão da arquitetura MVC.

1.2 ESTRUTURA DO TRABALHO

O texto está estruturado em quatro capítulos. No segundo capítulo é apresentada a fundamentação teórica utilizada para o desenvolvimento do trabalho. Trata da conceituação e utilização de geradores de código e da flexibilidade no uso da API JDBC para acesso a bancos de dados. Comenta sobre a utilidade dos *templates* e do *framework* Velocity, utilizado para a geração de código. A arquitetura MVC, adotada principalmente em aplicações web, o padrão de projeto DAO e a linguagem JSP também são descritos. O capítulo apresenta ainda algumas ferramentas com funcionalidades similares as da ferramenta descrita nesse trabalho.

No terceiro capítulo é apresentado o desenvolvimento da ferramenta, incluindo a especificação dos requisitos e dos casos de uso, a modelagem estrutural das classes, as ferramentas utilizadas no processo, a implementação e a operacionalidade da ferramenta

proposta.

Por último, no capítulo quatro, são apresentadas as conclusões e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nas seções seguintes são apresentados alguns aspectos teóricos relacionados ao trabalho, tais como: geradores de código, o uso da API JDBC para a extração do metadados de SGBDs relacionais, a utilização de *templates* e a sua manipulação através do motor de *templates* Velocity, o desenvolvimento de aplicações utilizando o padrão MVC, o acesso à base de dados utilizando o padrão de projeto DAO, a linguagem JSP e algumas ferramentas também utilizadas para a automatização do processo de desenvolvimento de software.

2.1 GERADORES DE CÓDIGO

Geradores de código são ferramentas que auxiliam no processo de desenvolvimento de software, proporcionando ganhos de produtividade e redução de tempo e de custos. A partir de uma entrada previamente definida, são gerados os arquivos fontes de uma aplicação. Essa saída pode servir tanto para o desenvolvimento completo de um sistema como somente para auxiliar na criação de rotinas específicas. Essas ferramentas são utilizadas principalmente quando se tem pouco tempo hábil ou uma pequena equipe para implementação do projeto.

Segundo Dalgarno (2006) e Herrington (2003, p. 3), a geração de código é a técnica pela qual se constrói código utilizando programas. Os geradores de código podem trabalhar em forma de linha de comando ou usar uma interface gráfica. Podem construir código para várias linguagens de programação, bem como efetuar a geração de uma única vez ou em etapas. As entradas e saídas são definidas conforme a necessidade, sendo que o desenvolvedor especifica o código de saída de forma manual.

A utilização de geradores de código traz benefícios substanciais ao desenvolvimento do projeto, aumentando a produtividade e a qualidade do software. As vantagens são alcançadas nos seguintes aspectos (HERRINGTON, 2003, p. 15-16):

a) qualidade: grandes quantidades de código escritas manualmente tendem a possuir uma baixa qualidade. A qualidade do código gerado está relacionada à qualidade do código definido nos *templates*. Quanto maior a qualidade do código utilizado pelo gerador, maior a qualidade do código resultante;
b) consistência: o código gerado pelo gerador de código tende a ser consistente. A

padronização da nomenclatura de classes, métodos e variáveis o torna de fácil entendimento e uso;

c) único ponto de conhecimento: caso se faça uso de vários geradores, ou de um gerador que tenha como finalidade a geração de código para várias camadas de uma aplicação, basta que o desenvolvedor conheça o ponto onde a alteração deverá ser efetuada. Uma vez alterada a entrada do gerador, ela será propagada para os processos conseguintes;

d) maior tempo para o projeto: o cronograma de um projeto que utiliza a geração de código é completamente diferente do cronograma de um projeto onde o desenvolvimento é totalmente manual. Em um projeto que adota o desenvolvimento manual, o uso inadequado de uma API pode ocasionar na reescrita de uma grande quantidade de código. Com o uso de geradores de código, os desenvolvedores podem modificar os *templates* para adequar-se a API e executar o gerador novamente, e todo o código gerado estará modificado contemplando as necessidades;

e) decisões atípicas do projeto: o alto nível das regras de negócio é perdido em minuciosas implementações de código. Como geradores de código utilizam arquivos pequenos, que especificam o "esqueleto" do código a ser gerado, as pequenas exceções são mais fáceis de serem tratadas, não sendo necessário avaliar um resultado de centenas de linhas de código;

f) abstração: os geradores de código podem construir códigos a partir de modelos abstratos do código alvo. Essa abstração permite que o código gerado possa ser facilmente migrado para outras linguagens e plataformas e a análise de negócio revista no escopo da abstração e não no código em execução;

g) agilidade no desenvolvimento: uma característica chave da geração de código é a maleabilidade do código de saída. No nível de negócio, o software será de mais fácil manutenção e implementação de novas funcionalidades ao longo do tempo.

Para Herrington (2003, p. 77), a construção de um gerador de código pode seguir as seguintes etapas de desenvolvimento:

a) identificação do que se deseja obter como saída do gerador;

b) definição de qual será a entrada e como a mesma será analisada;

c) interpretação e recuperação das informações da entrada, definindo a formatação e a geração da saída;

d) geração da saída a partir das informações extraídas do arquivo de entrada.

Stephens (2002) afirma que as possibilidades de geração de código são ilimitadas, mas não significa que deva ser utilizada em qualquer situação ou para qualquer projeto. É mais comum o uso de geradores de código em áreas específicas, incluindo geradores de interfaces, de acesso à base de dados, de documentação a partir de comentários presentes no código-fonte, de comandos para a linguagem SQL, de componentes e bibliotecas JSP.

2.2 API JDBC

Jeveaux (2004) afirma que efetuar uma comunicação direta com banco de dados é uma tarefa um tanto quanto complexa, pois existem muitos bancos de dados no mercado, e cada qual com sua linguagem proprietária.

> Algumas linguagens [de programação] são desenvolvidas para ter um alto desempenho, porém se tornam complexas para trabalhar com banco de dados. Outras linguagens têm um conjunto de primitivas que permite a criação de um banco de dados de forma fácil, mas se tornam deficientes em outros aspectos. (JEVEAUX, 2004).

A linguagem Java, mantendo as suas características, desenvolveu uma API para o acesso a banco de dados, que é totalmente independente de plataforma e fornecedor. A API JDBC facilita e padroniza a comunicação das aplicações Java com bancos de dados, oferecendo interfaces que possibilitam ao desenvolvedor criar seu próprio *driver* JDBC. Fabricantes de bancos de dados disponibilizam um *driver* JDBC completo, que se encarrega de implementar as classes que vão realizar as tarefas desejadas, bastando ao desenvolvedor apenas efetuar chamadas aos métodos das interfaces da API.

Segundo Deitel e Deitel (2005, p. 895), um *driver* JDBC permite que aplicativos Java conectem-se a um SGBD, possibilitando aos programadores manipular esse banco de dados utilizando os métodos definidos na API JDBC. Geralmente o JDBC é utilizado em conexões com bancos de dados relacionais. Entretanto, ele pode ser utilizado para acesso a qualquer origem de dados baseada em tabela. A API JDBC suporta quatro categorias de *drivers*, onde são implementadas formas diferentes de acesso aos bancos de dados (DEITEL; DEITEL, 2005, p. 908):

a) *driver* JDBC-ODBC (*driver* tipo 1): por padrão, as APIs do Java já incluem esse *driver* (`sun.jdbc.odbc.JdbcOdbcDriver`). Basicamente, ele efetua uma ponte entre programas Java e uma origem de dados ODBC, requerendo a instalação do

driver ODBC no computador onde a aplicação Java estiver executando;

b) *driver* de acesso nativo (*driver* tipo 2): permite que programas escritos em Java utilizem as APIs específicas do banco de dados, efetuando chamadas ao código nativo[3] do banco de dados. Essa comunicação é feita através da JNI, que efetua a ponte entre a JVM e o código nativo;

c) *driver* de acesso por *middleware* (*driver* tipo 3): esse *driver* é desenvolvido totalmente em Java, traduzindo as solicitações efetuadas pela aplicação em um protocolo de rede que não é específico do banco de dados. Ao receber essas solicitações, o servidor converte as informações para o protocolo proprietário do banco;

d) *driver* de acesso direto ao servidor (*driver* tipo 4): implementa protocolos de rede específicos do banco de dados, possibilitando aos programas Java a conexão direta à base de dados. Normalmente é desenvolvido pelo próprio fabricante do banco de dados.

Sobral (2004) afirma que para acessar um banco de dados a partir de um *driver* JDBC são necessários alguns passos, tais como: carregar e registrar o *driver* JDBC junto ao gerenciador de *driver*; configurar e obter uma conexão com o banco de dados; preparar a consulta; executar a consulta; obter e verificar os resultados; tratar possíveis erros; formatar a saída para o usuário; e fechar a conexão. Para efetuar esse processo, a API JDBC disponibiliza interfaces Java para acesso e manipulação do banco de dados, entre as quais tem-se:

a) `java.lang.Class`: essa classe não está contida no pacote `java.sql`, mas é através do método `forName(String param)` que o *driver* JDBC é instanciado e registrado na JVM;

b) `java.sql.Connection`: tem como função efetuar a conexão com o banco de dados e disponibilizar ao desenvolvedor as potencialidades dessa conexão, tais como a descrição das tabelas, a possibilidade de envio de instruções SQL ao banco de dados e a abertura e fechamento de conexões;

c) `java.sql.DriverManager`: efetua o gerenciamento dos *drivers* JDBC, efetuando a seleção do *driver* apropriado para a conexão com o banco de dados;

d) `java.sql.PreparedStatement`: armazena uma pré-compilação da instrução SQL

[3] Código nativo é o código escrito e compilado em uma linguagem específica de plataforma, como C ou C++ (DEITEL; DEITEL, 2005, p. 908).

a ser executada. É utilizado para a execução de instruções SQL no banco de dados, permitindo também a passagem de parâmetros para as instruções. É muito eficiente em instruções que necessitam ser executadas repetidamente;

e) `java.sql.Statement`: não guarda uma pré-compilação da instrução SQL, nem permite a passagem de parâmetros para a instrução, apenas efetua a execução de um comando SQL estático;

f) `java.sql.ResultSet`: é a representação, em forma de tabela, do resultado de uma instrução SQL executada no banco, permitindo a iteração entre as informações obtidas;

g) `java.sql.DataBaseMetadata`: armazena informações detalhadas da base de dados que são específicas de cada banco de dados. A partir dessa interface é possível obter informações, tais como a estrutura de tabelas e campos, as chaves primárias e as chaves estrangeiras.

2.3 BANCOS DE DADOS RELACIONAIS

Rumbaugh et al (1994, p. 486) define que o modelo de dados relacional se baseia no conceito de tabela, e um SGBD relacional é um programa de computador desenvolvido para o gerenciamento dessas tabelas. Para Deitel e Deitel (2005, p.896), um banco de dados relacional é uma representação lógica de dados, permitindo o acesso a esses dados sem considerar sua estrutura física, armazenando as informações em forma de tabelas (linhas e colunas).

Segundo Fagundes (2007), nos bancos de dados relacionais as tabelas são definidas tendo como base a teoria da normalização, que consiste em definir nas tabelas apenas os dados que sejam únicos para a entidade descrita, bem como definir relacionamentos para outras tabelas também normalizadas, evitando a redundância dos dados e facilitando as pesquisas e atualizações. Os bancos de dados relacionais estão fundamentados em uma forte teoria matemática e ferramentas bem desenvolvidas. Através da linguagem SQL é permitido que vários sistemas possam se comunicar entre si acessando o mesmo banco de dados.

Para Grayson (2002), um bom projeto de banco de dados em um modelo relacional possui qualidades como: reflexão estrutural real de um problema, representação de todos os dados esperados ao longo do tempo, armazenamento não redundante das informações, acesso

eficiente aos dados, manutenção da integridade dos dados e consistência das informações.

Atualmente existem dezenas de SGBDs relacionais no mercado, cada qual com sua particularidade, mas possuindo características comuns, que os define como sendo relacionais. Além do conceito de tabela, podem ser citadas como princípios chave de um SGBD relacional: a integridade de entidades e a integridade referencial. Rumbaugh et al (1994, p. 488) define como integridade de entidades a utilização de chave primária nas tabelas. Uma chave primária é uma combinação de um ou mais atributos (colunas), cujo valor deve ser único em cada registro (linha) da tabela. A integridade referencial é baseada nas chaves estrangeiras, que possuem a responsabilidade de manter as informações de uma tabela consistentes com a chave primária de outra tabela. Uma chave estrangeira também é chamada de relacionamento e é através dela que uma tabela referencia as informações contidas em outras tabelas.

Dentre a variedade de SGBDs existentes, podem ser citados o Oracle, o Microsoft SQL Server e o MySQL. Todos possuem o conceito de tabela, assim como o controle de integridade de entidades e de integridade referencial. Segundo Franco (2005), o Oracle XE é uma versão gratuita do Oracle 10g que pode ser distribuída juntamente com as aplicações para fins comerciais. O Microsoft SQL Server é um banco de dados comercial, robusto e usado por sistemas corporativos dos mais diversos portes. Funciona apenas sob algumas das várias versões dos sistemas operacionais da família Windows (MICROSOFT..., 2006). O MySQL, por sua vez, é um dos bancos de dados mais populares do mercado, sendo muito utilizado em aplicações web. No passado não possuía funcionalidades consideradas essenciais para um SGBD, mas atualmente já suporta recursos como *stored procedures*, *subselects*, integridade referencial e replicação de dados (MYSQL, 2007).

2.4 *TEMPLATES*

Takecian et al. (2004, p. 6) define *templates* como sendo arquivos que descrevem uma estrutura de formatação, que contém variáveis, blocos especiais e diretivas, indicando valores a serem inseridos e ações a serem executadas pelo sistema. Possuem a finalidade de gerar outros arquivos que contenham a formatação previamente definida no arquivo base. São muito utilizados em geração de código, pois o código de saída tem a tendência de seguir um padrão estabelecido.

A utilização de *templates* possibilita que a formatação do código gerado esteja externa ao código da aplicação que o gera. Dessa forma, caso exista a necessidade de alguma alteração na saída, não é preciso alterar a codificação do sistema, mas somente o arquivo utilizado como *template*.

Um *template* possui códigos estáticos e dinâmicos. Schvepe (2006, p. 11) define código dinâmico como sendo aquele que "passará por transformações no processo de geração do código de saída" e código estático com sendo aquele definido literalmente e que não sofrerá alterações pelo motor de *templates*.

2.5 MOTOR DE *TEMPLATES* VELOCITY

Motores de *templates* são mecanismos que viabilizam a separação do código dinâmico do código estático, possibilitando o desenvolvimento independente de ambas as partes. São muito utilizados no desenvolvimento de páginas web, geradores de código e geradores de documentação. Existem vários motores de *templates* que auxiliam de maneira rápida e fácil na construção de aplicações. Pode-se citar como exemplo o *Velocity Template Engine* (APACHE SOFTWARE FOUNDATION, 2004).

A atuação básica do Velocity consiste na formatação textual de dados, ou seja, na camada de apresentação de dados dos sistemas. O exemplo comum deste tipo de aplicação ocorre na geração dinâmica de páginas web. Devido à versatilidade de seu projeto, o Velocity pode também ser utilizado nas mais variadas aplicações, tais como, geração de código fonte SQL, XML e relatórios. Pode ser utilizado sozinho ou integrado com outras ferramentas de um sistema (MOURA; CRUZ, 2002).

Velocity não é um programa, mas um conjunto de classes Java (JEVEAUX; MOURA, 2002). Possui uma linguagem de manipulação de *templates*, a VTL, que permite fazer a inserção de informações de maneira dinâmica dentro do *template*. A VTL define as referências das variáveis no *template*; possui controles de fluxo de execução, como *loops* e comandos condicionais; permite a definição de macros e possui uma estrutura básica de substituição de valores de variáveis. O quadro 1 apresenta as principais estruturas sintáticas, incluindo comandos e diretivas da VTL.

ELEMENTO	SINTAXE	EXEMPLO

identificador	qualquer seqüência de caracteres alfabéticos, números, hífen ou *underline*, começando por um caracter alfabético	`id`
		`id123`
		`id_123`
		`id-123`

declaração de variáveis	identificador precedido por cifrão ($), identificador precedido por cifrão ($) e por exclamação (!), ou ainda identificador entre chaves precedido por cifrão ($)	`$id`
		`$!id`
		`${id}`

propriedades	seqüência de identificador, seguido de ponto (.), seguido de identificador, precedida por cifrão ($) ou precedida por cifrão ($) e exclamação (!), sendo que essa seqüência de termos pode estar ou não entre chaves	`$id.property`
		`$!id.property`
		`${id.property}`
		`$!{id.property}`

método	seqüência de identificador, ponto (.), identificador e abre e fecha parênteses, sendo que esta seqüência de termos pode estar ou não entre chaves, devendo ser precedida por um cifrão ($) ou por cifrão ($) e exclamação (!)	`$!id.metodo()`
		`$!{id.metodo(param1, param2)}`
		`$!{id.metodo($ref2, $ref2)}`
		`$id.metodo("param")`

atribuição	`#set($ref = valor)`	`#set($ref = "valor")`
		`#set($ref = id.property)`
		`#set($ref = id.metodo())`

controle de fluxo de execução	`#if (condição)`
	`  ...`
	`[#elseif (condição) ... ]*`
	`[#else ... ]`
	`#end`
	
	`#foreach($ref in $arg )`
	`  ...`
	`#end`
		`#if ($x > 1)`
		`  ...`
		`#end`
		
		`#foreach ($x in $list)`
		`  ...`
		`#end`

definição de macros	`#macro`
	` (nomeDaMacro$arg1[,$arg]*)`
	`  < código VTL >`
	`#end`
		`#macro(linhas $lista)`
		`  #foreach( $item in $lista )`
		`  <tr><td $item </td></tr>`
		`  #end`
		`#end`

invocação de macros	`#nomeDaMacro($ref)`	`#set($cor = "red")` `#nomeDaMacro($cor)`

inclusão de arquivos	`#include`: permite incluir arquivos que não serão analisados pelo Velocity
	
	`#parse`: permite incluir *templates* que serão analisados pelo Velocity
		`#include("texto.txt")`
		
		`#parse("templateAuxiliar.vm")`

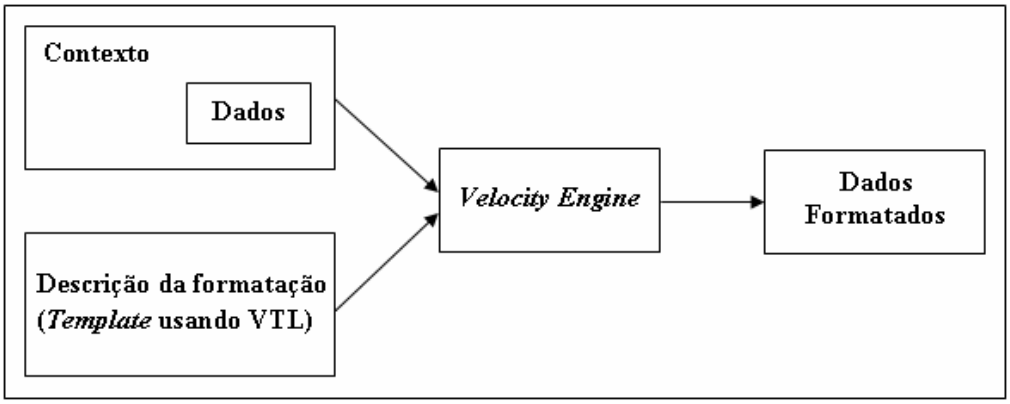
comentários	podem ser de linha, quando iniciados com ##, ou de bloco, quando iniciados com #* e finalizados com *#	`## comentário em uma linha`
		
		`#*`
		` comentário em várias linhas`
		`*#`

Fonte: adaptado de Moura e Cruz (2002).

Quadro 1 – Elementos da linguagem VTL

Segundo Moura e Cruz (2002), os dados formatados na forma textual são gerados a

partir dos dados brutos e de uma descrição da formatação. Os dados brutos devem estar encapsulados em classes Java, sendo acessados por meio de uma interface pública (métodos públicos). O Velocity infere, em tempo de execução, esta interface pública e através dela pode realizar a recuperação dos dados armazenados nas classes. A figura 1 mostra como está organizada a arquitetura do Velocity.



Fonte: adaptado de Moura e Cruz (2002).

Figura 1 – Arquitetura do motor de *templates* Velocity

Durante uma execução normal do Velocity, são realizados os seguintes passos (MOURA; CRUZ, 2002):

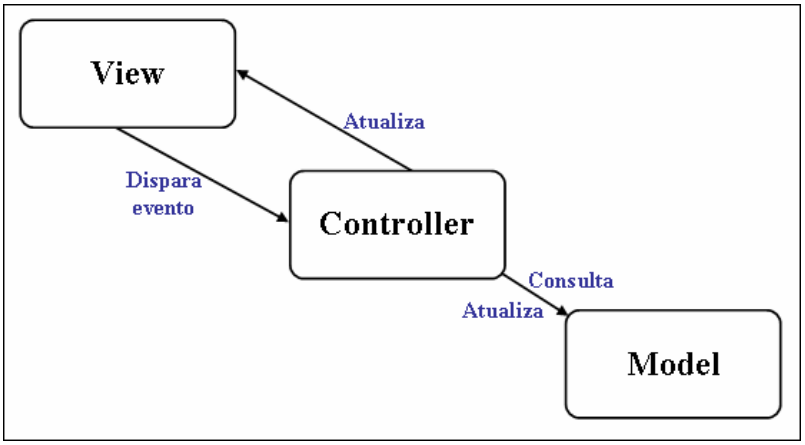
a) inicialização do Velocity, carregando uma configuração que definirá algumas características de seu funcionamento;
b) criação do contexto, que é o mapeamento entre as referências no arquivo de formatação em VTL e as classes Java;
c) obtenção do *template* especificado;
d) análise do *template* e armazenamento do mesmo em memória, na forma de uma estrutura de dados conhecida como AST;
e) realização da substituição adequada das referências pelos valores obtidos dos objetos Java, de acordo com as regras de mapeamento definidas pelo contexto.

Em uma aplicação web, a utilização do Velocity faz com que o código Java fique totalmente separado do código HTML, tornando assim a aplicação muito mais modularizada e de fácil manutenção. Possibilita o trabalho independente e paralelo, trazendo maior produtividade no desenvolvimento de aplicações. Desenvolvedores deixam a codificação das interfaces por conta dos *designers*, e esses, por sua vez, não precisam se preocupar com detalhes complexos da linguagem de programação.

2.6 MVC

O MVC é um padrão de arquitetura de software, geralmente usado em padrões de projeto de software, embora o seu conceito compreenda mais da arquitetura de uma aplicação do que um típico padrão de projeto (MVC, 2007). O padrão MVC organiza a interatividade de uma aplicação dentro de três camadas distintas. A primeira camada *(Model*) representa os dados e a lógica de negócio da aplicação, a segunda (*View*) provê a visualização dos dados e a interação do usuário com a aplicação, e a terceira camada (*Controller*) efetua o gerenciamento do fluxo de execução da aplicação através do controle das requisições (SUN MICROSYSTEMS, 2002a).

Segundo Bezerra (2002, p. 246), uma camada de software é uma coleção de unidades de software que podem ser executadas ou acessadas. A divisão de um software em camadas permite que este seja portável e modificável. As alterações efetuadas em uma das camadas não afetam as demais, desde que o mecanismo de comunicação entre elas permaneça inalterado. A figura 2 mostra a comunicação das camadas no padrão MVC.



Fonte: adaptado de Sun Microsystems (2002b).

Figura 2 – Padrão MVC

A camada de modelo é a representação dos dados e das regras de negócio da aplicação, sendo responsável por gerenciar o acesso e manutenção dos dados. Ela é uma camada independente, não referenciando as camadas de visão e de controle. Sua função é disponibilizar serviços e dados para as outras camadas da aplicação. A camada de visão renderiza o conteúdo da camada de modelo, obtendo os dados e os apresentando ao usuário através de uma interface gráfica ou textual. A camada de visão geralmente tem acesso à camada de modelo, mas não pode alterar o seu estado. A camada de controle é responsável por coordenar as atividades entre as camadas de modelo e visão. Ela faz chamadas aos

métodos do modelo para poder alterar as informações da base de dados. Após alterar o estado do modelo, a camada de controle atualiza a camada de visão com as informações que foram alteradas (SUN MICROSYSTEMS, 2002a).

Quando fala-se no conceito de camadas, segundo Sun Microsystems (2002a), freqüentemente são usados os termos MVC 1 e MVC 2. Essa terminologia descreve dois padrões básicos para o desenvolvimento de páginas JSP. MVC 1 e MVC 2 referem-se à ausência ou presença, respectivamente, de *servlets* como controladores de requisições provenientes da camada de visão.

A arquitetura do modelo MVC 1 consiste em um *browser* acessando diretamente páginas JSP. Essas, por sua vez, acessam um componente JavaBeans[4] que representa o modelo da aplicação e a seleção de visões é determinada por *hyperlinks* dentro do código do documento ou por uma requisição com passagem de parâmetros. Nesse modelo, o controle é descentralizado, pois a página corrente é que determina qual será a próxima página a ser apresentada. Também são as próprias páginas JSP que processam suas requisições através do envio de parâmetros via *get* ou *post.*

O modelo MVC 2 introduz a utilização de *servlets* como controladores entre o *browser* e as páginas JSP. O controlador é um meio de seleção de visões, desacoplando as páginas JSP do código dos *servlets*. Ele centraliza a lógica de seleção das visões de acordo com as requisições, passagens de parâmetros ou o estado da aplicação. Esse modelo possibilita que as aplicações tenham uma melhor manutenção e extensão, provendo um único ponto de controle.

2.7 DAO

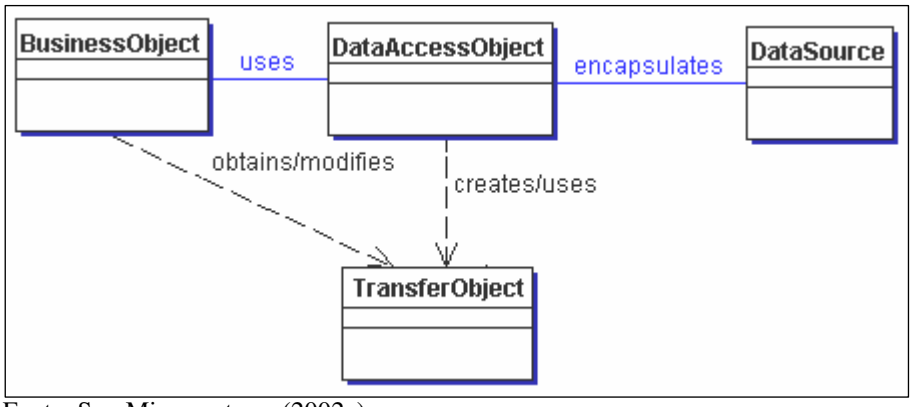
O padrão DAO descreve o desenvolvimento de um mecanismo independente de acesso aos dados, permitindo que o acesso ao meio de armazenamento dos dados esteja separado das regras de negócio da aplicação. É um modelo de implementação de uma camada de abstração que efetua o acesso às informações armazenadas em uma base de dados.

Encapsulando todo o código de acesso ao meio de armazenamento, o DAO gerencia a conexão com a fonte de dados e obtém as informações persistidas na base de dados. O DAO

[4] JavaBeans são componentes de software projetados para serem unidades reutilizáveis, que uma vez criados podem ser reusados sem modificação de código e em qualquer propósito de aplicação, seja um *applet*, um s*ervlet* ou qualquer outra (TELEMACO; LIMA, 2004).

também implementa todo o código de manipulação das informações e externaliza esses serviços aos objetos de negócio. Esses, por sua vez, utilizam os serviços disponibilizados, mas não conhecem como eles estão implementados. Essa abstração permite ao DAO adaptar-se a diferentes meios de armazenamento de dados sem que seus clientes, ou componentes de negócio, sejam afetados (SUN MICROSYSTEMS, 2002c).

A figura 3 mostra o diagrama de classes que representa o padrão DAO.



Fonte: Sun Microsystems (2002c).

Figura 3 – Padrão DAO

Uma classe de negócio usa a classe DAO para a obtenção de informações da base de dados. O DAO, que encapsula a implementação do acesso à base de dados, recupera a informação solicitada e a transfere para a classe de negócio. A classe DAO também pode receber, por transferência, alguma informação da classe de negócio para que as alterações sejam efetuadas na base de dados.

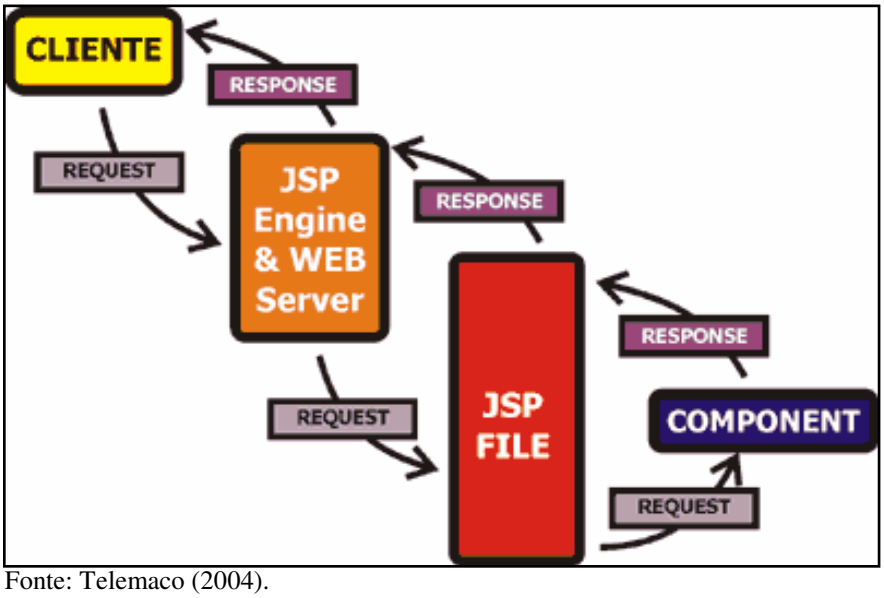
2.8 JSP

Segundo Telemaco (2005), JSP é uma tecnologia para desenvolvimento de aplicações web, que possui a vantagem da portabilidade de plataforma, podendo ser executada em outros sistemas operacionais. Ela permite ao desenvolvedor de *sites* produzir aplicações com recursos como: acesso a banco de dados, acesso a arquivos texto, captação de informações a partir de formulários e o uso de variáveis e *loops*. JSP oferece também a vantagem de ser facilmente codificada, facilitando assim a elaboração e manutenção de uma aplicação. Essa tecnologia permite separar a programação lógica (parte dinâmica) da programação visual (parte estática), facilitando o desenvolvimento de aplicações mais robustas, onde programador e *designer* possam trabalhar no mesmo projeto, mas de forma independente.

Menin (2005, p. 16-17) descreve que os arquivos JSP são arquivos de texto armazenados no servidor e seu conteúdo é formado por código Java e código HTML.

> Os arquivos JSPs são arquivos de texto armazenados no servidor, normalmente com a extensão .jsp que substituem as páginas HTML tradicionais. Os arquivos JSP contêm HTML tradicional com código embutido para permitir que aplicações Java sejam executadas no servidor. Quando uma página JSP é solicitada, o código é processado no servidor e o conteúdo dinâmico é obtido juntamente com o conteúdo HTML que é enviado ao cliente que solicitou a página. A primeira vez que uma página JSP é acessada, ela é automaticamente transformada em um *servlet* pelo servidor, através do mecanismo JSP, sendo que este *servlet* é executado nos próximos acessos à página. (MENIN, 2005, p. 16-17).

O processamento de uma página JSP é mostrado na figura 4. As solicitações a um arquivo JSP são feitas por um cliente através do *browser*, e essa solicitação é enviada ao servidor web através de um *request*. O servidor web, ou JSP *Engine*, envia a solicitação de qualquer componente (JavaBeans, *Servlet* ou EJB) especificado no arquivo. O componente controla a requisição possibilitando a recuperação de informações em um banco de dados ou outro dado armazenado. O componente retorna o resultado da requisição ao servidor web, através de um *response*. O servidor web envia a página JSP revisada de volta para o cliente e o usuário visualiza os resultados através do seu *browser*. Nesse processo, o protocolo de comunicação usado entre o cliente e o servidor pode ser HTTP ou outro.



Fonte: Telemaco (2004).

Figura 4 – Processamento de uma solicitação a uma página JSP

Deitel e Deitel (2005, p. 960-961) mostram uma visão geral sobre JSP, descrevendo que são quatro os componentes chaves:

a) diretivas: são as mensagens enviadas ao contêiner JSP, encarregado de executar o código JSP. Permitem ao desenvolvedor especificar as configurações das páginas, incluindo conteúdo de outros recursos e especificando bibliotecas de *tags* personalizadas;

b) ações: encapsulam funcionalidades de *tags* predefinidas pelo programador;
c) elementos de *script*: permitem aos desenvolvedores inserir código Java que interaja com componentes em uma página JSP;
d) bibliotecas de *tags*: permitem que os desenvolvedores de páginas web manipulem o conteúdo do código JSP sem conhecimento prévio da linguagem Java.

2.9 TRABALHOS CORRELATOS

Existem diversas aplicações com funcionalidades semelhantes as da ferramenta proposta nesse trabalho, cada uma com suas particularidades, seja na implementação ou no código gerado. A ferramenta apresentada nesse trabalho é fortemente baseada no CodGer, uma aplicação que efetua a geração de páginas JSP a partir de uma base de dados em MySQL. Outras ferramentas que apresentam funções similares são: EasyCase, TechCodeGenerator e ClassGenerator.

2.9.1 CodGer

Menin (2005) descreve uma ferramenta para geração de código que tem como objetivo auxiliar analistas e programadores na construção de aplicativos na linguagem JSP a partir das definições de uma base de dados em MySQL. Com o auxílio do software, é possível a construção de interfaces de cadastro sem a necessidade de efetuar processos repetitivos e dispendiosos.

As páginas JSP são geradas após três procedimentos: conexão e leitura da estrutura da base de dados, configuração das telas a serem geradas e geração de código de saída. É feita a leitura do dicionário de dados e as informações são armazenadas em memória. Logo após, o usuário determina quais informações estarão disponíveis na tela, selecionando as tabelas, atributos e relacionamentos desejados. Feito isso, é dado início ao processo de geração de código que, por sua vez, cria as páginas de cadastro.

A ferramenta define o código gerado dentro do seu código fonte, realiza a geração de código seguindo o padrão MVC e as páginas geradas possuem funcionalidades de inclusão, alteração, exclusão e consulta. No seu desenvolvimento foram utilizados o ambiente de

programação Eclipse, o servidor de aplicações JBoss, o banco de dados MySQL e a linguagem Java.

2.9.2 EasyCase

EasyCase é um protótipo de uma ferramenta CASE que efetua a geração de código na linguagem Java. Por possuir uma integração com o dicionário de dados do Oracle Designer6i, em seu desenvolvimento não foi adotada a metodologia de orientação a objetos. Isso se fez necessário para que fossem mantidas as características do dicionário de dados usado na ferramenta da Oracle (SANTOS, 2005, p. 36). Essa ferramenta possibilita a criação e manutenção de um dicionário de dados para bancos de dados relacionais, gerando, a partir do dicionário especificado, interfaces para a manipulação das informações através das funções básicas de inserção, alteração, exclusão e consulta.

O modelo físico é armazenado em um formato específico e as interfaces geradas utilizam componentes da API Swing. Foi desenvolvida no padrão MVC, dividindo a manipulação dos dados, a interatividade com o usuário e a lógica de negócio em módulos distintos. EasyCase permite o mapeamento das coleções do Oracle Designer6i para o seu dicionário de dados, podendo ser empregada na migração de tecnologias. Dessa forma, empresas que utilizam o Oracle Designer6i podem facilmente migrar suas aplicações para a tecnologia Java, sem a necessidade de alterações do metadados existente.

A ferramenta possui geradores de código para a criação da base de dados e das interfaces de usuário e ambas são geradas a partir do dicionário especificado pelo desenvolvedor. O gerador de SQL é responsável por gerar os comandos apropriados para criar fisicamente tabelas, índices e restrições, de acordo com a especificação de um banco de dados relacional. Já o gerador de interfaces cria os arquivos fonte em Java, instancia o ambiente de execução da JVM e abre a conexão com o banco de dados, disponibilizando ao usuário a interface com as operações de inserção, alteração, exclusão e consulta dos dados.

2.9.3 TechCodeGenerator

TechCodeGenerator foi criada com o objetivo de reduzir as etapas repetitivas no processo de desenvolvimento de software, permitindo ao desenvolvedor uma atenção maior

ao código que implementa efetivamente as rotinas do sistema (TECHNIQUE TI, 2003). Através da junção de rotinas, como o mapeamento do banco de dados e a codificação de telas, são criadas interfaces com métodos de inclusão, alteração, exclusão e consulta.

A ferramenta efetua uma engenharia reversa da estrutura física do banco de dados, minimizando o esforço de especificação, restando ao desenvolvedor configurar as informações ainda não definidas. A partir dos dados obtidos, é gerada uma classe de persistência que efetua os comandos *insert*, *update*, *delete* e *select*. O usuário tem a liberdade de escolher os campos que farão parte da interface, assim como as regras sobre cada um deles.

TechCodeGenerator efetua a geração de código utilizando a arquitetura MVC, sendo geradas classes Java para as camadas de modelo e controle e código XSL para a camada de apresentação. A extração do metadados é portável, podendo ler dicionários de dados de bancos de dados como Oracle, MySQL, Microsoft SQL Server, Sybase e DB2.

2.9.4 ClassGenerator

Segundo Leal (2005, p. 12), ClassGenerator é uma ferramenta que tem o objetivo de otimizar diversos aspectos das fases de projeto, codificação e testes, combinando bons conceitos das práticas e técnicas bem sucedidas no desenvolvimento de software. A partir de um esquema de banco de dados, ClassGenerator gera um conjunto de artefatos como classes de negócio, classes básicas, interfaces com usuário e documentação do código. As classes de negócio possuem a responsabilidade do mapeamento objeto-relacional e regras de negócio, as classes básicas possuem um construtor público e métodos *getter* e *setter*, e as interfaces exercem as funcionalidades de inclusão, alteração, exclusão e consulta. Por fim, a documentação é gerada no estilo PHPDoc e JavaDoc.

A leitura do dicionário de dados é feita considerando que o projeto de banco de dados esteja bem definido, normalizado e livre de erros. Para cada tabela do esquema é gerada uma nova classe básica, e para cada uma dessas classes são criadas outras para o mapeamento do banco de dados e interação com a interface. As classes podem ser geradas nas linguagens PHP, Java e Phyton. Já as interfaces são geradas na linguagem XHTML, seguindo alguns padrões como: (i) arquitetura estabelecida pelo gerador, mas adaptável para arquiteturas do tipo MVC; (ii) utilização de *templates*; e (iii) utilização de folhas de estilos CSS.

3 DESENVOLVIMENTO DO TRABALHO

Este capítulo apresenta o desenvolvimento da ferramenta FormGenerate, descrevendo as tecnologias e ferramentas utilizadas na sua implementação. Na elaboração e construção de FormGenerate foram seguidas as seguintes etapas:

a) especificação dos requisitos funcionais e não funcionais: foram identificados e detalhados;
b) definição dos bancos de dados: foram elencados três bancos de dados relacionais (Oracle XE, Microsoft SQL Server 2000, MySQL 5.0.21) para a ferramenta efetuar a leitura do dicionário de dados e gerar código;
c) estudo da API JDBC: utilizando os *drivers* JDBC específicos de cada banco de dados elencado, foi verificada a forma como é realizada a extração das informações do metadados, conforme sugerido por Herrington (2003) no 2º passo do desenvolvimento de geradores de código;
d) interpretação das informações de entrada: foi definido como as informações de entrada, obtidas através da extração do metadados do banco de dados, seriam armazenadas e formatadas para a posterior geração de código (3º passo sugerido por Herrington (2003));
e) especificação da saída: foi implementado manualmente o código de saída a ser gerado pela ferramenta (1º passo sugerido por Herrington (2003));
f) definição do gerenciador de páginas: foi definido o Tomcat como gerenciador de aplicações;
g) especificação da ferramenta: foi especificada com análise orientada a objeto utilizando a UML, para o desenvolvimento dos diagramas de caso de uso, de atividades, de pacotes e de classes;
h) implementação: foi efetuada considerando os requisitos especificados, as informações extraídas do metadados dos bancos de dados e o código de saída a ser gerado, conforme 4º passo do desenvolvimento de geradores de código sugerido por Herrington (2003);
i) elaboração dos *templates*: foram criados para a geração do código de conexão com o banco de dados e para as camadas de modelo, controle e visão;
j) definição de uma base de dados para testes: foi modelada e posteriormente criada em cada banco de dados especificado, sendo geradas as páginas JSP a partir das

configurações definidas na ferramenta.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A atividade de levantamento dos requisitos[5] corresponde à etapa de compreensão do problema aplicada ao desenvolvimento de software. A seguir são apresentados os requisitos não funcionais e funcionais atendidos pela ferramenta. No quadro 2 são relacionados os requisitos não funcionais. O quadro 3 apresenta os requisitos funcionais e sua rastreabilidade, ou seja, vinculação com os casos de uso associados.

REQUISITOS NÃO FUNCIONAIS
RNF001 – Gerar páginas na linguagem JSP, seguindo o padrão da arquitetura MVC.
RNF002 – Ser implementado na linguagem Java 5.0.
RNF003 – Ser implementado utilizando o ambiente de desenvolvimento Eclipse 3.2.
RNF004 – Utilizar *templates* para a geração do código de saída.
RNF005 – Utilizar o motor de *templates* Velocity para formatação dos *templates*.
RNF006 – Utilizar a API JDBC para a extração das informações do metadados dos SGBDs.
RNF007 – Utilizar *drivers* JDBCs específicos para cada SGDB definido.
RNF008 – Gerar os arquivos de configuração no formato XML.

Quadro 2 – Requisitos não funcionais

[5] O principal objetivo do levantamento de requisitos é fazer com que usuários e desenvolvedores tenham a mesma visão do problema a ser resolvido (BEZERRA, 2002, p. 20).

REQUISITOS FUNCIONAIS	CASO DE USO
RFDEFPRJ001 – Criar diferentes projetos.	UC001
RFDEFPRJ002 – Salvar as definições de um projeto.	UC001
RFDEFPRJ003 – Abrir um projeto previamente salvo.	UC001
RFDEFPRJ004 – Permitir configurar o local onde o projeto será salvo.	UC001
RFDEFPRJ005 – Criar diretórios para salvar os formulários, grupos e relatórios.	UC001
RFDEFPRJ006 – Informar os *templates* que serão utilizados para a geração dos arquivos das camadas de modelo, controle e visão.	UC003
RFDEFPRJ007 – Criar diretórios para armazenar os arquivos gerados para as camadas de modelo, controle e visão.	UC001
RFDEFPRJ008 – Identificar os *templates* utilizados para a geração de código.	UC003
RFCONBD001 – Configurar as informações de conexão com o banco de dados.	UC002
RFCONBD002 – Conectar com diferentes SGBDs.	UC002
RFCONBD003 – Ler as informações do metadados e armazená-las em memória.	UC002
RFCONBD004 – Salvar as informações de conexão com o banco de dados.	UC002
RFDEFFRM001 – Criar formulários a partir do metadados extraído.	UC004
RFDEFFRM002 – Permitir configurar os formulários.	UC004
RFDEFFRM003 – Salvar as definições dos formulários.	UC004
RFDEFFRM004 – Abrir um formulário previamente salvo.	UC004
RFDEFGRP001 – Criar grupos de formulários previamente cadastrados.	UC005
RFDEFGRP002 – Permitir configurar os grupos.	UC005
RFDEFGRP003 – Salvar as definições dos grupos de formulários.	UC005
RFDEFGRP004 – Abrir um grupo previamente salvo.	UC005
RFDEFREL001 – Criar relatórios para a visualização das informações da base de dados.	UC006
RFDEFREL002 – Permitir configurar os relatórios.	UC006
RFDEFREL003 – Salvar as definições dos relatórios.	UC006
RFDEFREL004 – Abrir um relatório previamente salvo.	UC006
RFGERCOD001 – Gerar formulários com funcionalidades de inclusão, alteração, exclusão e consulta.	UC007
RFGERCOD002 – Gerar código para a camada de modelo.	UC007
RFGERCOD003 – Gerar código para a camada de controle.	UC007
RFGERCOD004 – Gerar código para a camada de visão.	UC007

Quadro 3 – Requisitos funcionais

3.2 ESPECIFICAÇÃO

Segundo Guedes (2005, p. 6), a UML é uma linguagem visual utilizada para modelar sistemas computacionais. Utilizando a UML através da ferramenta Enterprise Architect, foi especificada a ferramenta através dos seguintes diagramas: casos de uso, atividades, pacotes e classes.

3.2.1 Diagrama de casos de uso

A figura 5 apresenta o diagrama de casos[6] de uso da ferramenta desenvolvida.

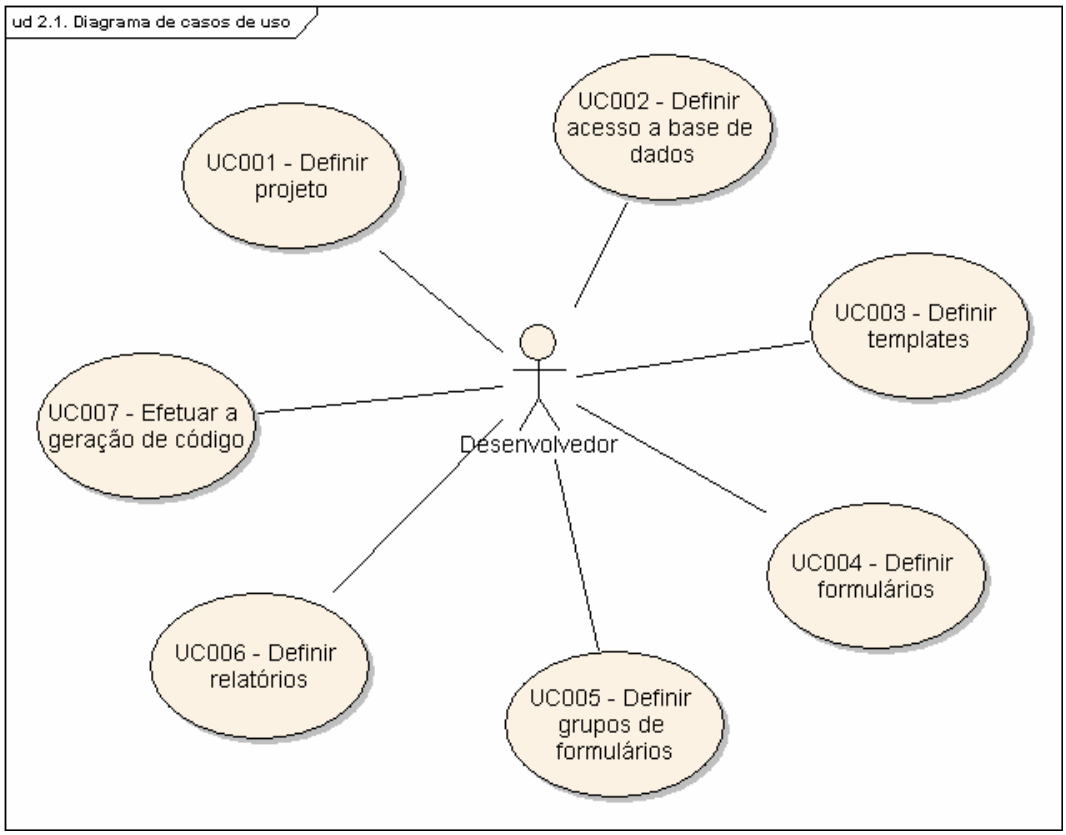


Figura 5 – Diagrama de casos de uso

Os quadros 4 a 10 apresentam a descrição dos casos de uso da ferramenta.

[6] Este é o diagrama mais geral e informal da UML, sendo utilizando normalmente nas fases de levantamento e análise de requisitos do sistema. Apresenta uma linguagem simples e de fácil compreensão para que os usuários possam ter uma idéia geral de como o sistema irá se comportar (GUEDES, 2005, p. 7). Segundo Bezerra (2002, p. 45), o diagrama de casos de uso é o modelo de especificação dos requisitos e direciona as tarefas posteriores do ciclo de vida do software, mostrando as possíveis iterações dos atores quando do uso da ferramenta; as etapas que devem ser executadas pelo ator e pelo sistema para que o caso de uso execute sua função; as restrições e validações que o caso de uso deve possuir.

Definir projeto
Sumário: O desenvolvedor efetua as operações referentes à definição do projeto, criando novos projetos, alterando e salvando as suas configurações.
Ator primário: Desenvolvedor.
Fluxo principal: Criar novo projeto 1) O usuário acessa o menu correspondente ao projeto. 2) É apresentada a tela correspondente às definições do projeto. 3) O usuário clica no botão "Novo projeto". 4) Os campos da tela são habilitados para digitação. 5) O usuário preenche as configurações do projeto. 6) O usuário clica no botão "Salvar configurações do projeto". 7) A ferramenta verifica se todas as informações foram preenchidas corretamente. 8) A ferramenta cria os diretórios padrões do projeto. 9) A ferramenta salva as informações do projeto em um arquivo XML. 10) A ferramenta apresenta a mensagem "Configurações do projeto salvas com sucesso!!!".
Fluxo alternativo: Abrir projeto existente No passo 2, caso o usuário opte por abrir um projeto previamente salvo: 2.1) O usuário clica no botão "Abrir projeto". 2.2) O usuário seleciona o projeto desejado. 2.3) Os campos da tela são habilitados e carregados com as definições do projeto. 2.2) Retorna ao passo 4.
Fluxo de exceção: Validar informações No passo 7, caso o usuário deixe de digitar alguma informação: 7.1) A ferramenta verifica qual informação não foi informada. 7.2) A ferramenta apresenta uma mensagem contendo qual informação precisa ser digitada. 7.3) Retorna ao passo 5.
Fluxo de exceção: Consistir arquivo de projeto No passo 2.2, caso o usuário selecione um arquivo que não corresponde a um arquivo de definição do projeto: 2.2.1) A ferramenta exibe uma mensagem informando que o arquivo selecionado é inválido. 2.2.2) Retorna ao passo 2.
Pós-condições: Os diretórios do projeto foram criados. Um arquivo XML contendo as definições do projeto foi criado.

Quadro 4 – Descrição do caso de uso UC001 – Definir projeto

Definir acesso à base de dados
Sumário: O desenvolvedor efetua a configuração para acesso ao banco de dados, informando o banco de dados a ser utilizado, o servidor onde o banco está instalado, a base de dados, o *login* e a senha para conexão. Pode também salvar essas configurações nas definições do projeto.
Ator primário: Desenvolvedor.
Pré-condições: Um novo projeto deve ter sido criado ou um projeto existente deve ter sido aberto.
Fluxo principal: Configurar acesso ao banco de dados 1) O usuário acessa o menu correspondente às configurações de acesso ao banco de dados. 2) É apresentada a tela correspondente às configurações de acesso ao banco de dados. 3) O usuário clica no botão "Configurar conexão". 4) Os campos da tela são habilitados para digitação. 5) O usuário preenche as configurações de acesso ao banco de dados. 6) O usuário clica no botão "Salvar configurações de conexão". 7) A ferramenta verifica se todas as informações foram preenchidas corretamente. 8) A ferramenta salva as informações de acesso ao banco de dados no arquivo de definições do projeto. 9) A ferramenta apresenta a mensagem "Configurações de conexão com o Banco de Dados salvas com sucesso!!!".
Fluxo alternativo: Conectar com o banco de dados No passo 5, após configurar o acesso ao banco de dados: 5.1) O usuário clica no botão "Conectar com a base de dados". 5.2) A ferramenta valida a conexão com o banco de dados. 5.3) A ferramenta exibe a mensagem "Conexão efetuada com sucesso!!!". 5.4) Retorna ao passo 6.
Fluxo alternativo: Extrair metadados Após o passo 5.3, caso o usuário opte por carregar as informações da base de dados: 5.3.1) O usuário clica no botão "Carregar base de dados". 5.3.2) A ferramenta exibe a mensagem "Base de dados carregada com sucesso!!!". 5.3.3) Retorna ao passo 6.
Fluxo de exceção: Validar conexão com o banco de dados No passo 5.2, caso as informações de conexão estejam incorretas: 5.2.1) A ferramenta trata o erro ocorrido. 5.2.1) A ferramenta apresenta uma mensagem correspondente ao erro. 5.2.2) Retorna ao passo 5.
Fluxo de exceção: Erro ao carregar base de dados No passo 5.3.1, caso ocorra um erro ao carregar as informações da base de dados: 5.3.1.1) A ferramenta trata o erro ocorrido. 5.3.1.2) A ferramenta apresenta uma mensagem correspondente ao erro. 5.3.1.3) Retorna ao passo 5.
Fluxo de exceção: Validar informações No passo 7, caso o usuário deixe de digitar alguma informação: 7.1) A ferramenta verifica qual informação não foi informada. 7.2) A ferramenta apresenta uma mensagem contendo qual informação precisa ser digitada. 7.3) Retorna ao passo 5.
Pós-condições: As configurações de conexão com o banco de dados foram salvas no mesmo arquivo de definição do projeto.

Quadro 5 – Descrição do caso de uso UC002 – Definir acesso ao banco de dados

Definir *templates*
Sumário: O desenvolvedor informa quais *templates* serão utilizados para a geração de código. Pode também salvar essas configurações nas definições do projeto.
Ator primário: Desenvolvedor.
Pré-condições: Um novo projeto deve ter sido criado ou um projeto existente deve ter sido aberto.
Fluxo principal: Definir *templates* 1) O usuário acessa o menu correspondente às configurações dos *templates*. 2) É apresentada a tela correspondente às configurações dos *templates*. 3) O usuário clica no botão "Configurar *templates*". 4) Os campos da tela são habilitados para digitação. 5) O usuário informa os arquivos *templates*. 6) O usuário clica no botão "Salvar *templates*". 7) A ferramenta verifica se todas as informações foram preenchidas corretamente. 8) A ferramenta salva as informações dos *templates* no arquivo de configuração do projeto. 9) A ferramenta copia os *templates* para uma pasta dentro do projeto. 10) A ferramenta apresenta a mensagem "Configurações dos *templates* salvas com sucesso!!!".
Fluxo de exceção: *Template* duplicado No passo 7, caso o usuário informe o mesmo *template* para duas finalidades: 7.1) A ferramenta identifica a duplicidade. 7.2) A ferramenta apresenta uma mensagem informando que não é possível definir o mesmo *template* para duas finalidades. 7.3) Retorna ao passo 5.
Fluxo de exceção: Validar informações No passo 7, caso o usuário deixe de digitar alguma informação: 7.1) A ferramenta verifica qual informação não foi informada. 7.2) A ferramenta apresenta uma mensagem contendo qual informação precisa ser digitada. 7.3) Retorna ao passo 5.
Pós-condições: As configurações dos *templates* foram salvas no mesmo arquivo de definição do projeto. Os *templates* foram copiados para uma pasta padrão do projeto.

Quadro 6 – Descrição do caso de uso UC003 – Definir *templates*

Definir formulários
Sumário: O desenvolvedor cria formulários que serão utilizados na geração de código da ferramenta. Um formulário representa uma tela do sistema a ser gerado automaticamente. As configurações de um formulário são salvas em um arquivo, e uma referência do formulário é armazenada no arquivo de definição do projeto.
Ator primário: Desenvolvedor.
Pré-condições: Um novo projeto deve ter sido criado ou um projeto existente deve ter sido aberto. A conexão com o banco de dados deve estar estabelecida e a base de dados carregada.
Fluxo principal: Criar novo formulário 1) O usuário acessa o menu correspondente às configurações de formulário. 2) É apresentada a tela correspondente às configurações de formulário. 3) O usuário clica no botão "Novo formulário". 4) Os campos da tela são habilitados para digitação. 5) A ferramenta carrega para a tela as tabelas do banco de dados disponíveis para seleção. 6) O usuário configura o formulário. 7) O usuário clica no botão "Salvar formulário". 8) A ferramenta verifica se todas as informações foram preenchidas corretamente. 9) A ferramenta salva as configurações do formulário em um arquivo XML, em uma pasta padrão do projeto, e guarda uma referência do formulário no arquivo de configuração do projeto. 10) A ferramenta apresenta a mensagem "Configurações do formulário salvas com sucesso!!!".
Fluxo alternativo: Abrir formulário No passo 2, caso o usuário opte por abrir um formulário previamente salvo: 2.1) O usuário clica no botão "Abrir formulário". 2.2) Os campos da tela são habilitados e carregados com as definições do formulário. 2.3) Retorna ao passo 4.
Fluxo alternativo: Excluir formulário Após o passo 2.2, caso o usuário opte por excluir o formulário: 2.2.1) O usuário clica no botão "Excluir formulário". 2.2.2) A ferramenta exclui o arquivo contendo as configurações do formulário. 2.2.3) A ferramenta remove a referência do formulário das configurações do projeto. 2.2.4) Retorna ao passo 2.
Fluxo de exceção: Validar informações No passo 8, caso o usuário deixe de digitar alguma informação: 8.1) A ferramenta verifica qual informação não foi informada. 8.2) A ferramenta apresenta uma mensagem contendo qual informação precisa ser digitada. 8.3) Retorna ao passo 6.
Pós-condições: Um arquivo XML contendo as definições do formulário foi salvo. Uma referência do formulário foi salva no arquivo de definições do projeto.

Quadro 7 – Descrição do caso de uso UC004 – Definir formulários

Definir grupos de formulários
Sumário: O desenvolvedor cria grupos de formulários que serão utilizados na geração de código da ferramenta. Um grupo representa um item de menu na tela principal do sistema a ser gerado automaticamente. As configurações de um grupo são salvas em um arquivo, e uma referência do grupo é armazenada no arquivo de definição do projeto.
Ator primário: Desenvolvedor.
Pré-condições: Um novo projeto deve ter sido criado ou um projeto existente deve ter sido aberto. A conexão com o banco de dados deve estar estabelecida e a base de dados carregada. Ao menos um formulário deve ter sido criado.
Fluxo principal: Criar novo grupo 1) O usuário acessa o menu correspondente às configurações de grupo. 2) É apresentada a tela correspondente às configurações de grupo. 3) O usuário clica no botão "Novo grupo". 4) Os campos da tela são habilitados para digitação. 5) A ferramenta carrega para a tela os formulários disponíveis para seleção. 6) O usuário configura o grupo. 7) O usuário clica no botão "Salvar grupo". 8) A ferramenta verifica se todas as informações foram preenchidas corretamente. 9) A ferramenta salva as configurações do grupo em um arquivo XML, em uma pasta padrão do projeto, e guarda uma referência do grupo no arquivo de configuração do projeto. 10) A ferramenta apresenta a mensagem "Configurações do grupo salvas com sucesso!!!".
Fluxo alternativo: Abrir grupo No passo 2, caso o usuário opte por abrir um grupo previamente salvo: 2.1) O usuário clica no botão "Abrir grupo". 2.2) Os campos da tela são habilitados e carregados com as definições do grupo. 2.3) Retorna ao passo 4.
Fluxo alternativo: Excluir grupo Após o passo 2.2, caso o usuário opte por excluir o grupo: 2.2.1) O usuário clica no botão "Excluir grupo". 2.2.2) A ferramenta exclui o arquivo contendo as configurações do grupo. 2.2.3) A ferramenta remove a referência do grupo das configurações do projeto. 2.2.4) Retorna ao passo 2.
Fluxo de exceção: Validar informações No passo 8, caso o usuário deixe de digitar alguma informação: 8.1) A ferramenta verifica qual informação não foi informada. 8.2) A ferramenta apresenta uma mensagem contendo qual informação precisa ser digitada. 8.3) Retorna ao passo 6.
Pós-condições: Um arquivo XML contendo as definições do grupo foi salvo. Uma referência do grupo foi salva no arquivo de definições do projeto.

Quadro 8 – Descrição do caso de uso UC005 – Definir grupos de formulários

Definir relatórios
Sumário: O desenvolvedor cria relatórios que serão utilizados na geração de código da ferramenta. As configurações de um relatório são salvas em um arquivo, e uma referência do relatório é armazenada no arquivo de definição do projeto.
Ator primário: Desenvolvedor.
Pré-condições: Um novo projeto deve ter sido criado ou um projeto existente deve ter sido aberto. A conexão com o banco de dados deve estar estabelecida e a base de dados carregada. Ao menos um formulário deve ter sido criado.
Fluxo principal: Criar novo relatório 1) O usuário acessa o menu correspondente às configurações de relatório. 2) É apresentada a tela correspondente às configurações de relatório. 3) O usuário clica no botão "Novo relatório". 4) Os campos da tela são habilitados para digitação. 5) A ferramenta carrega para a tela os formulários disponíveis para seleção. 6) O usuário configura o relatório. 7) O usuário clica no botão "Salvar relatório". 8) A ferramenta verifica se todas as informações foram preenchidas corretamente. 9) A ferramenta salva as configurações do relatório em um arquivo XML, em uma pasta padrão do projeto, e guarda uma referência do relatório no arquivo de configuração do projeto. 10) A ferramenta apresenta a mensagem "Configurações do relatório salvas com sucesso!!!".
Fluxo alternativo: Abrir relatório No passo 2, caso o usuário opte por abrir um relatório previamente salvo: 2.1) O usuário clica no botão "Abrir relatório". 2.2) Os campos da tela são habilitados e carregados com as definições do relatório. 2.3) Retorna ao passo 4.
Fluxo alternativo: Excluir relatório Após o passo 2.2, caso o usuário opte por excluir o relatório: 2.2.1) O usuário clica no botão "Excluir relatório". 2.2.2) A ferramenta exclui o arquivo contendo as configurações do relatório. 2.2.3) A ferramenta remove a referência do relatório das configurações do projeto. 2.2.4) Retorna ao passo 2.
Fluxo de exceção: Validar informações No passo 8, caso o usuário deixe de digitar alguma informação: 8.1) A ferramenta verifica qual informação não foi informada. 8.2) A ferramenta apresenta uma mensagem contendo qual informação precisa ser digitada. 8.3) Retorna ao passo 6.
Pós-condições: Um arquivo XML contendo as definições do grupo foi salvo Uma referência do grupo foi salva no arquivo de definições do projeto.

Quadro 9 – Descrição do caso de uso UC006 – Definir relatórios

Efetuar a geração de código
Sumário: O desenvolvedor efetua a geração de código para as camadas de modelo, controle e visão. O desenvolvedor tem a opção de gerar código para todas as camadas de uma única vez, ou gerar cada camada individualmente.
Ator primário: Desenvolvedor.
Pré-condições: Um novo projeto deve ter sido criado ou um projeto existente deve ter sido aberto. A conexão com o banco de dados deve estar estabelecida e a base de dados carregada. Ao menos um formulário deve ter sido criado. Ao menos um grupo deve ter sido criado.
Fluxo principal: Gerar código 1) O usuário acessa o menu correspondente a geração de código. 2) O usuário clica no botão "Gerar todas as camadas". 3) A ferramenta efetua a geração de código para todas as camadas, apresentando uma mensagem ao término da geração de cada uma delas.
Fluxo alternativo: Gerar camadas independentes No passo 1, caso o usuário opte por gerar as camadas individualmente: 1.1) O usuário clica em um dos botões: "Gerar camada de modelo", "Gerar camada de controle" ou "Gerar camada de visão". 1.2) A ferramenta efetua a geração de código da respectiva camada. 1.3) A ferramenta apresenta uma mensagem informando que a geração de código da camada foi efetuada com sucesso.
Fluxo alternativo: Erro na geração de código No passo 3, caso ocorra um erro no processo de geração de código: 3.1) A ferramenta trata o erro ocorrido. 3.2) A ferramenta apresenta uma mensagem correspondente ao erro ocorrido. 3.3) Retorna ao passo 1.
Pós-condições: A geração de código para as camadas de modelo, controle e visão foi efetuada.

Quadro 10 – Descrição do caso de uso UC007 – Efetuar geração de código

3.2.2 Diagrama de atividades

A figura 6 descreve o processo principal desempenhado pela ferramenta, através do diagrama de atividades[7], desde a criação, ou abertura, do projeto até a geração de código, passando pelas fases de configuração do projeto, configuração de acesso à base de dados, seleção dos *templates* e criação de formulários, grupos e relatórios.

[7] O diagrama de atividade é um diagrama orientado a fluxo de controle, sendo um tipo especial do diagrama de estados, em que são representados os estados de uma atividade (BEZERRA, 2002, p. 228). Para Guedes (2005, p. 11), o diagrama de atividades preocupa-se em descrever os passos a serem percorridos para a conclusão de uma atividade específica, podendo representar um método com certo grau de complexidade ou até mesmo modelar um processo completo.

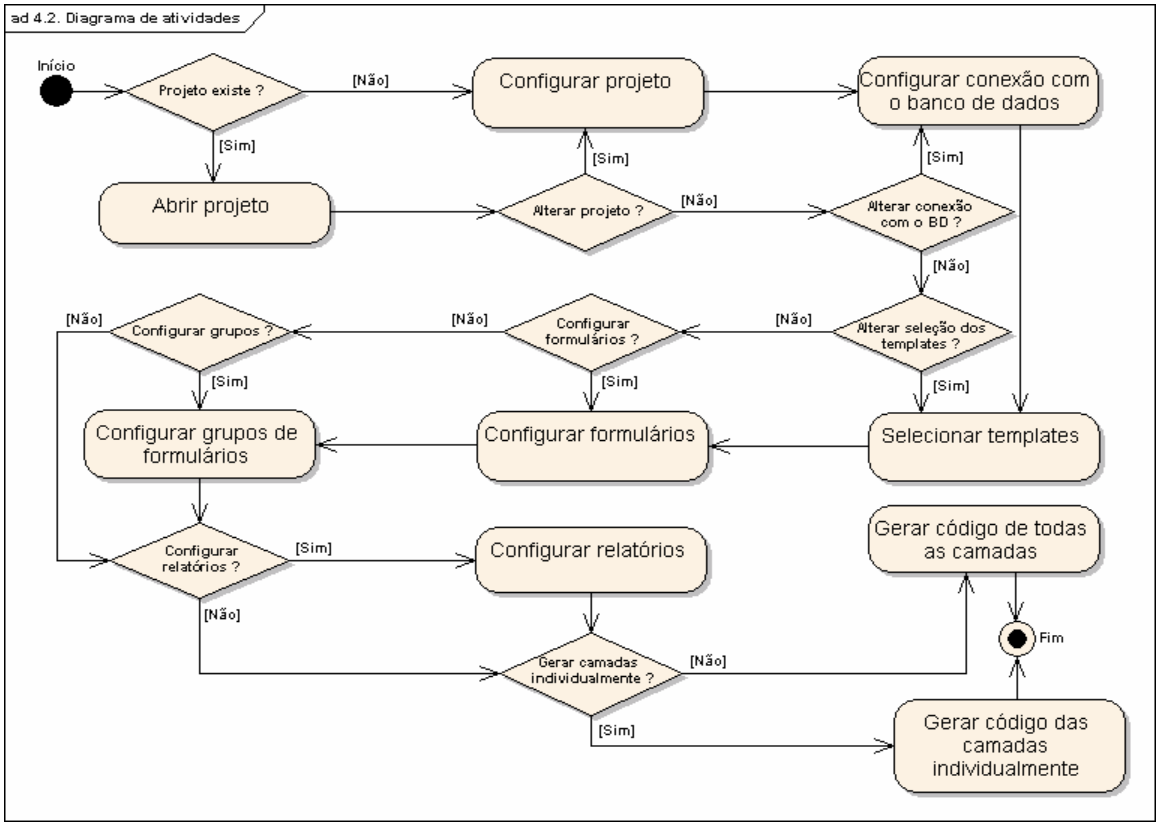


Figura 6 – Diagrama de atividades

Para iniciar a configuração das páginas a serem geradas, é necessário ter um projeto em memória, sendo que o mesmo pode ser carregado de duas formas: criando um novo projeto ou abrindo um projeto previamente salvo.

Se o desenvolvedor optar por criar um novo projeto, o mesmo terá que iniciar todas as configurações necessárias: informar os dados do projeto, configurar a conexão com o banco de dados, extrair as definições do metadados e, por fim, informar os arquivos *templates* que serão utilizados na geração de código.

Com a primeira parte das configurações efetuada e o metadados extraído, o desenvolvedor pode iniciar a configuração dos formulários, grupos e relatórios. Nesse processo, o desenvolvedor pode configurar as suas páginas conforme for identificando as suas necessidades. A configuração dos relatórios é opcional, mas para que a geração de código seja feita é necessário que exista ao menos um formulário e um grupo criados. Tendo todas as definições de formulários, grupos e relatórios criadas, é possível processar a geração de código de duas formas: (i) gerando cada camada individualmente, ou (ii) gerando as camadas de modelo, controle e visão de uma única vez.

Caso o desenvolvedor abra um projeto já existente, ao carregá-lo em memória será permitido efetuar a manutenção de qualquer configuração (projeto, banco de dados, *templates*, formulários, grupos e relatórios) e também será possível processar a geração das páginas JSP.

3.2.3 Diagrama de pacotes

A figura 7 mostra o diagrama de pacotes[8] da aplicação.

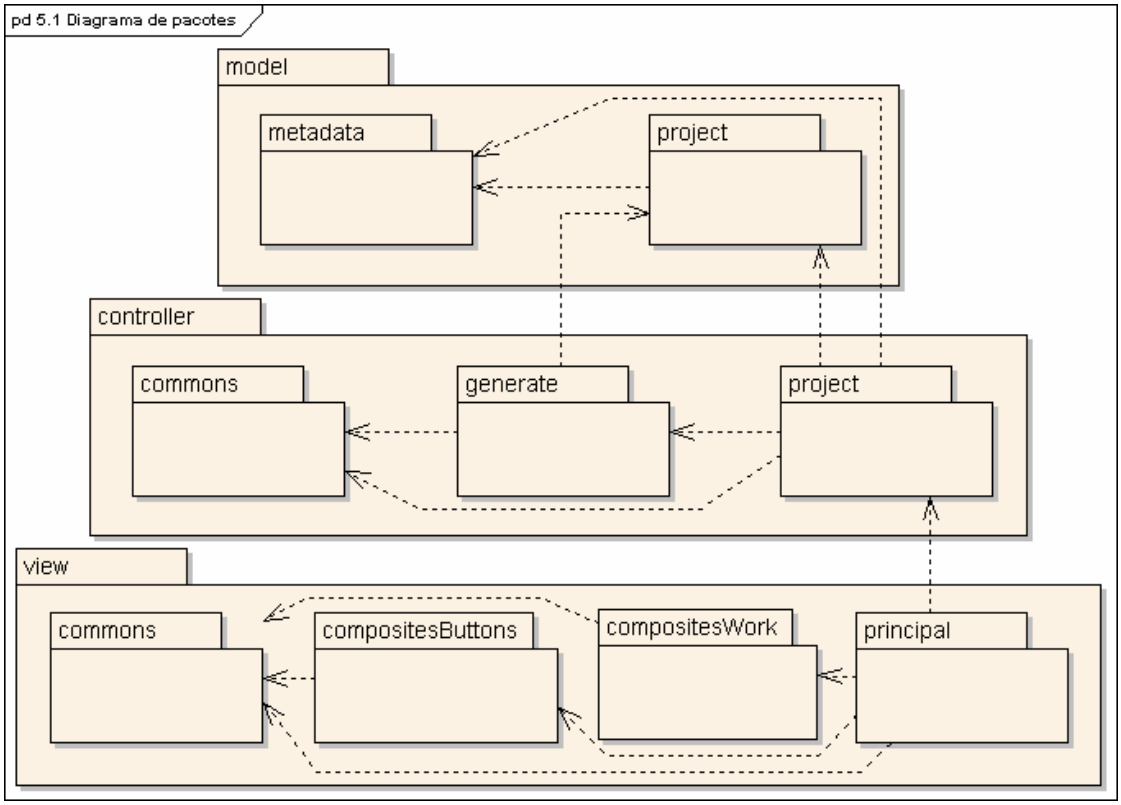


Figura 7 – Diagrama de pacotes

Basicamente, a ferramenta foi dividida em três pacotes distintos: `model`, `controller` e `view`. O pacote `model` agrega as classes responsáveis por extrair as informações do metadados e mantê-las em memória, e armazenar as informações necessárias para a geração de código. O pacote `controller` é o responsável por executar as requisições vindas da camada de visão, consultando e alterando o estado dos objetos da camada de modelo. Por fim, o pacote `view` possui as classes que definem a interface com o usuário e que efetuam a comunicação com a camada de controle.

A figura 8 mostra o diagrama de pacotes que demonstra a comunicação entre as camadas na aplicação. A intenção desse diagrama não é denotar todos os pacotes e classes da aplicação, mas sim, exibir como está definida a arquitetura da ferramenta. O diagrama

[8] Segundo Guedes (2005, p. 14), o diagrama de pacotes tem por objetivo representar os subsistemas ou sub-módulos englobados por um sistema de forma a determinar as partes que o compõem, podendo ser utilizado de maneira independente ou associado com outros diagramas. Bezzera (2002, p. 236-237) define pacotes como sendo um agrupador de classes, mas que também pode ser utilizado como um mecanismo de agrupamento geral, podendo agrupar vários artefatos de um modelo. Um pacote pode conter relacionamentos de dependência com outros pacotes, o que permite a construção de um diagrama de pacotes apresentando essas dependências.

apresenta os pacotes `model`, `controller` e `view`, definindo explicitamente o modelo definido pelo padrão MVC, e as classes que se relacionam entre esses pacotes.

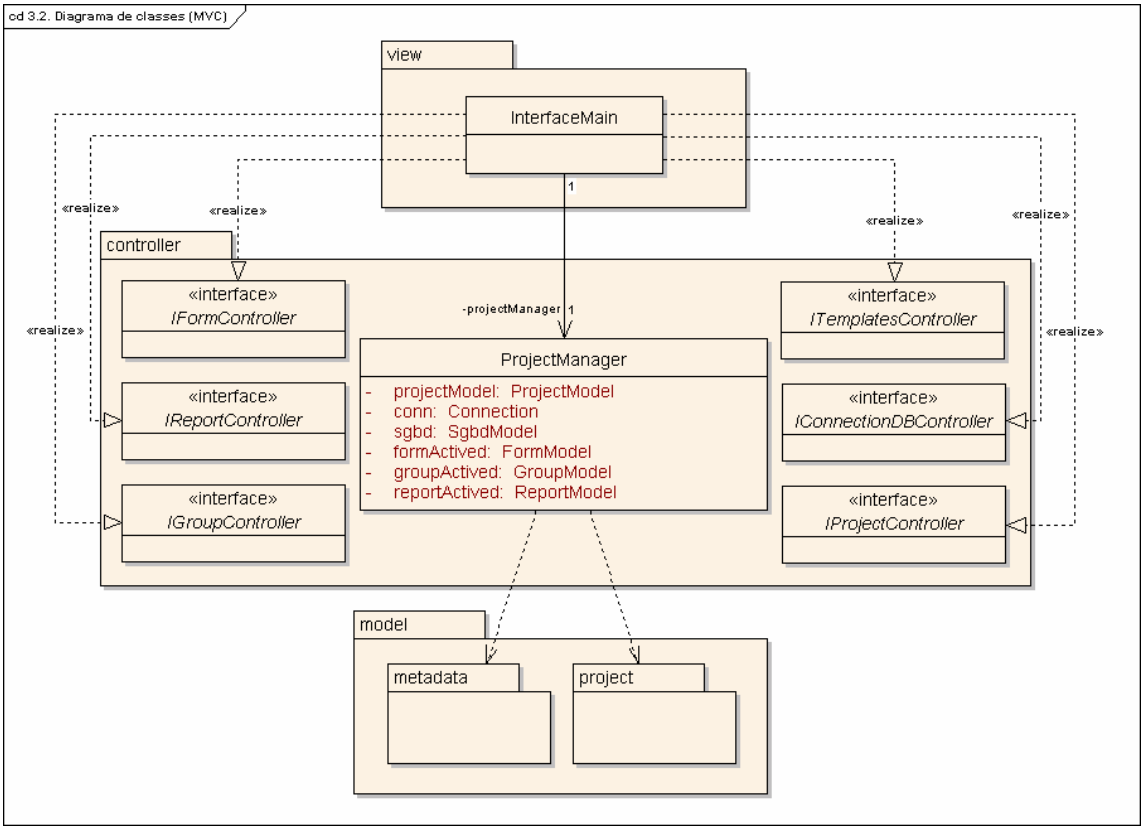


Figura 8 – Diagrama de pacotes descrevendo o padrão MVC aplicado na ferramenta

A classe principal do pacote `view` é a `InterfaceMain`, que agrega todos os componentes e telas da GUI e implementa as interfaces definidas na camada de controle. Ela acessa a camada de controle através de um atributo que armazena a referência para um objeto da classe `ProjectManager`.

A classe `ProjectManager` se encarrega de consultar e atualizar a camada de modelo e devolver à camada de visão o resultado das requisições. Essa comunicação é efetuada através das interfaces implementadas pela classe `InterfaceMain`. Já a comunicação com a camada de modelo é feita através do acesso às classes dos pacotes `model.metadata` e `model.project`.

No pacote `model.metadata`, a classe `LoadDatabase` é responsável pela extração do metadados da base de dados. Cada método `static` dessa classe recupera um tipo de objeto do banco de dados, sendo que os objetos tratados são: tabelas (*Table*), colunas (*Columns*), chaves primárias (*PrimayKey*) e chaves estrangeiras (*ForeignKey*). No pacote `model.project` estão todas as classes responsáveis por armazenar as informações referentes aos formulários, grupos e relatórios configurados pelo desenvolvedor.

3.2.4 Diagrama de classes

A figura 9 mostra o diagrama de classes[9] do pacote `model.metadata`. Foram definidos métodos *getters* e *setters* para os atributos de cada classe, mas os mesmos não estão sendo apresentados no diagrama. Em `model.metadata` estão armazenadas as classes responsáveis por efetuar a conexão com o banco de dados, obter as informações do metadados da base de dados e fazer o mapeamento dos objetos do banco, armazenando os mesmos em memória. Todas as classes desse pacote implementam a interface `java.io.Serializable`, permitindo que os objetos possam ser salvos em disco.

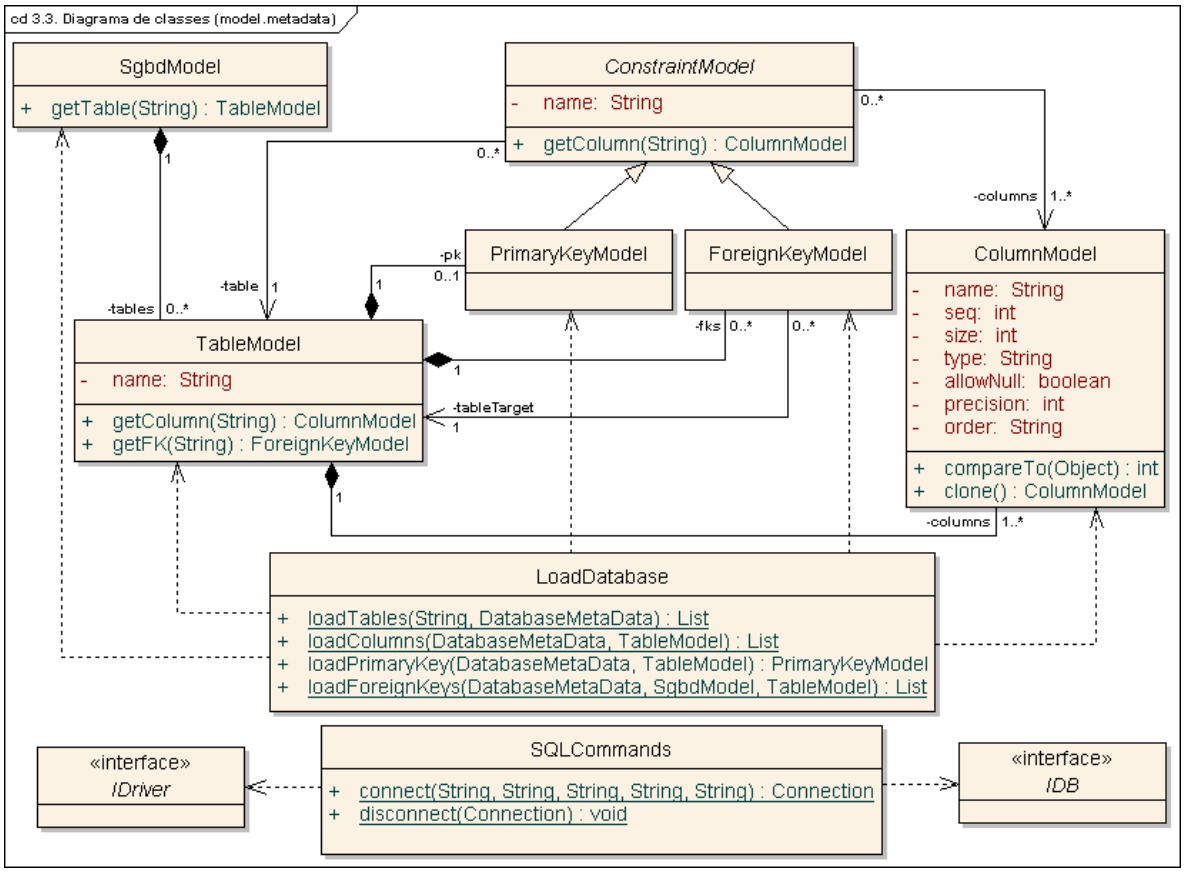


Figura 9 – Diagrama de classes do pacote `model.metadata`

Segue o detalhamento das classes relacionadas no diagrama da figura 9, descrevendo o papel de cada uma delas na estrutura:

a) `SgbdModel`: classe responsável por armazenar a lista de tabelas extraída do

[9] Guedes (2005, p. 8) descreve que o diagrama de classes é o diagrama mais utilizado e o mais importante da UML, servindo de apoio para a maioria dos outros diagramas. Ele define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos de cada classe, estabelecendo também o relacionamento entre elas. Segundo Bezerra (2002, p. 96), o modelo de classes evolui durante as iterações do desenvolvimento do sistema sendo incrementado à medida que o sistema é desenvolvido.

metadados através da API JDBC. É instanciada na criação de um novo projeto;

b) `TableModel`: classe responsável por armazenar uma tabela extraída do metadados através da API JDBC. Objetos dessa classe armazenam as informações de uma tabela do banco de dados;

c) `ColumnModel`: classe responsável por armazenar uma coluna de uma tabela extraída do metadados através da API JDBC. Objetos dessa classe armazenam as informações de uma coluna de uma tabela do banco de dados;

d) `ConstraintModel`: classe abstrata responsável por armazenar as informações das *Constraints* extraídas do metadados através da API JDBC;

e) `PrimaryKeyModel`: especialização da classe `ConstraintModel`. É responsável por armazenar as informações da *PrimaryKey* de uma tabela extraída do metadados através da API JDBC. Objetos dessa classe armazenam as informações da chave primária de uma tabela do banco de dados;

f) `ForeignKeyModel`: especialização da classe `ConstraintModel`. É responsável por armazenar as informações da *ForeignKey* de uma tabela extraída do metadados através da API JDBC. Objetos dessa classe armazenam as informações da chave estrangeira de uma tabela do banco de dados;

g) `LoadDatabase`: classe responsável por recuperar as informações da estrutura da base de dados para a memória;

h) `SQLCommands`: classe responsável por selecionar o *driver* JDBC específico de cada banco de dados, montar a URL de conexão e efetuar a conexão com o banco de dados;

i) `IDB`: interface que define atributos `static` identificando os SGBDs utilizados;

j) `IDriver`: interface que define atributos `static` identificando os *drivers* JDBC dos SGBDs utilizados.

Em `model.project` estão armazenadas as classes que definem as informações referentes ao projeto, configuração de acesso ao banco de dados, configuração dos *templates*, formulários, grupos e relatórios. Todas as classes desse pacote implementam a interface `java.io.Serializable`, permitindo que os objetos possam ser salvos em disco. A figura 10 mostra o diagrama de classes do pacote `model.project`. Foram definidos métodos *getters* e *setters* para os atributos de cada classe, mas os mesmos não estão sendo apresentados no diagrama.

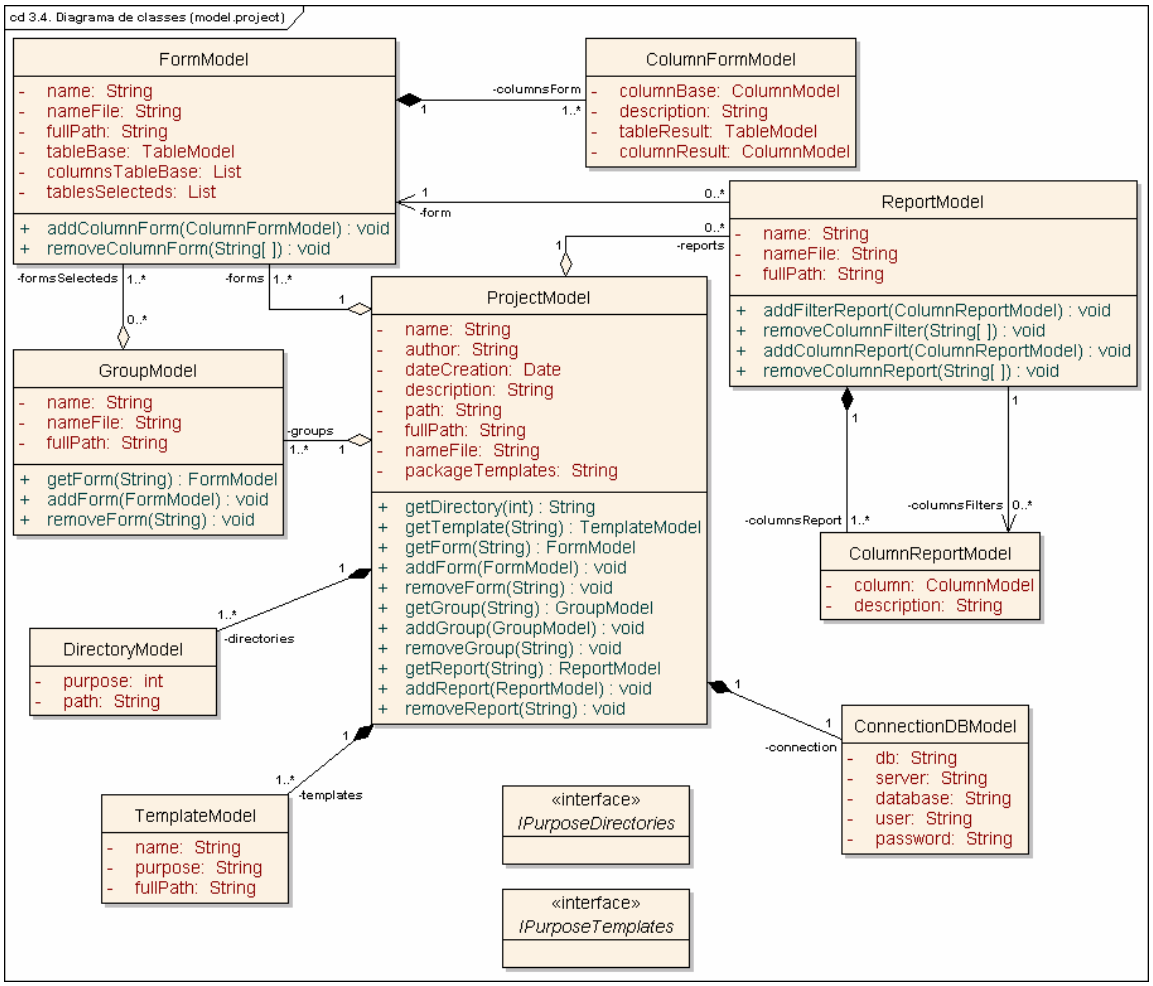


Figura 10 – Diagrama de classes do pacote `model.project`

Segue o detalhamento das classes relacionadas no diagrama da figura 10, descrevendo o papel de cada uma delas na estrutura:

a) `ProjectModel`: classe responsável por armazenar as informações de um projeto criado pela ferramenta. É instanciada na criação de um novo projeto;

b) `DirectoryModel`: classe responsável por armazenar o caminho de cada diretório do projeto. Cada diretório é representado por um objeto, e os mesmos são instanciados na criação de um novo projeto;

c) `ConnectionDBModel`: classe responsável por armazenar as informações de conexão com o banco de dados. É instanciada no momento em que as configurações de acesso ao banco de dados são salvas;

d) `TemplateModel`: classe responsável por armazenar as informações de cada *template* informado no projeto. Cada *template* é representado por um objeto, e os mesmos são instanciados quando as configurações dos *templates* são salvas;

e) `FormModel`: classe responsável por armazenar as configurações de um formulário. Objetos dessa classe são instanciados quando um novo formulário é criado;

f) `ColumnFormModel`: classe responsável por armazenar as configurações das colunas de um formulário. Objetos dessa classe são instanciados ao inserir uma nova coluna no formulário;

g) `GroupModel`: classe responsável por armazenar as informações de um grupo de formulários. Objetos dessa classe são instanciados quando um novo grupo de formulários é criado;

h) `ReportModel`: classe responsável por armazenar as informações de um relatório. Objetos dessa classe são instanciados quando um novo relatório é criado;

i) `ColumnReportModel`: classe responsável por armazenar as colunas e filtros de um relatório. Objetos dessa classe são instanciados ao inserir uma nova coluna ou um novo filtro no relatório;

j) `IPurposeDirectories`: interface que define atributos `static` identificando a finalidade dos diretórios do projeto;

k) `IPurposeTemplates`: interface que define atributos `static` identificando a finalidade dos *templates* utilizados na geração de código.

O pacote `controller` especifica interfaces que disponibilizam os serviços da camada de controle. Também é responsável por executar todas as requisições vindas da camada de visão e por atualizar e consultar as informações da camada de modelo. A figura 11 apresenta o diagrama de classes do pacote `controller`.

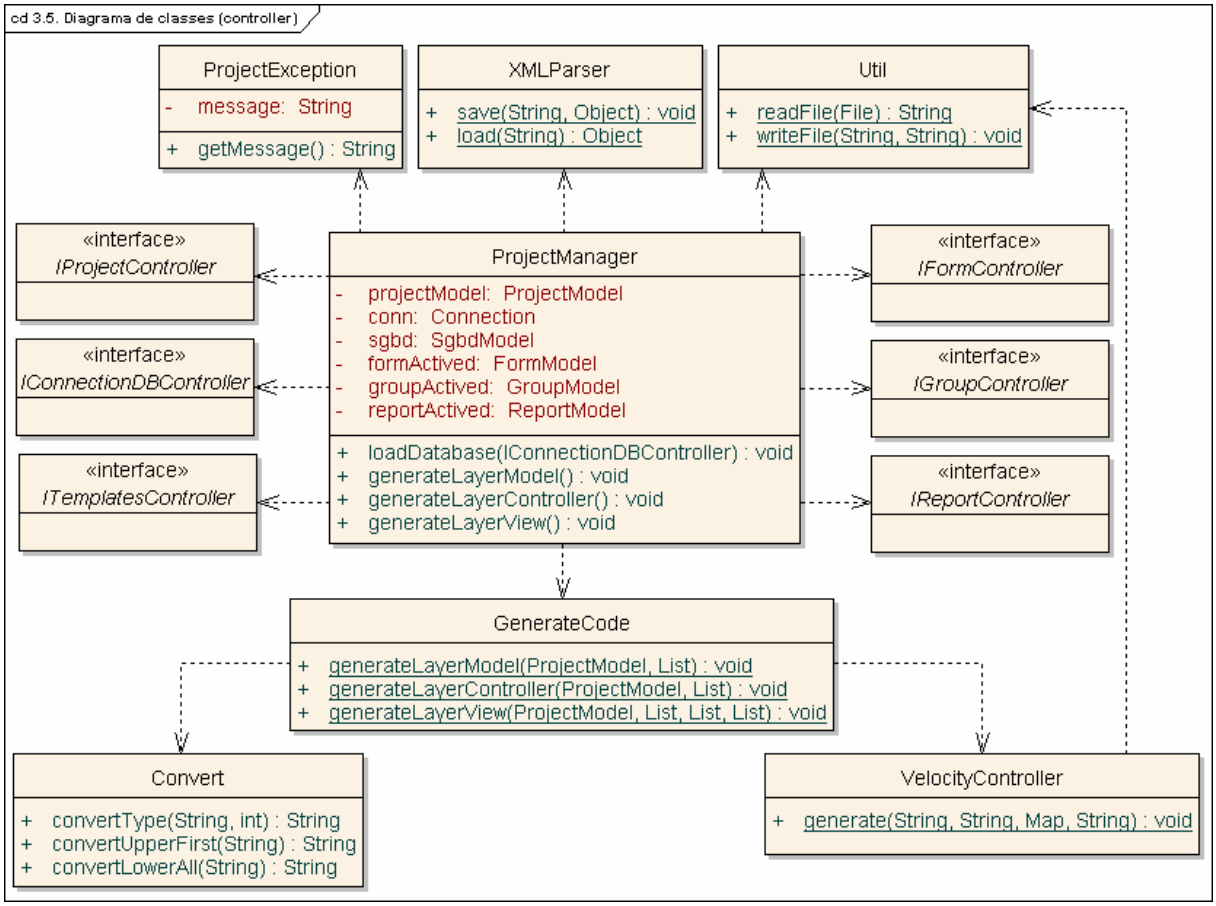


Figura 11 – Diagrama de classes do pacote `controller`

Segue o detalhamento das classes relacionadas no diagrama da figura 11, descrevendo o papel de cada uma delas na estrutura:

a) `ProjectManager`: classe que gerencia as requisições vindas da camada de visão e efetua a comunicação com a camada de modelo. Ela é responsável por efetuar todas as operações necessárias para a criação e manutenção dos objetos que armazenam as informações necessárias para a geração de código;

b) `IProjectController`: interface que define os métodos necessários para a criação e manutenção de um projeto;

c) `IConnectionDBController`: interface que define os métodos necessários para a configuração das informações de conexão com a base de dados;

d) `ITemplateController`: interface que define os métodos necessários para a configuração dos *templates* utilizados para a geração de código;

e) `IFormController`: interface que define os métodos necessários para a criação e manutenção de um formulário;

f) `IGroupController`: interface que define os métodos necessários para a criação e manutenção de um grupo de formulários;

g) `IReportController`: interface que define os métodos necessários para a criação e manutenção de um relatório;

h) `XMLParser`: classe responsável por gravar os objetos em disco e recuperar as informações desses objetos;

i) `Util`: classe responsável por efetuar a gravação e leitura de arquivos em disco;

j) `ProjectException`: classe responsável pelo tratamento de exceções provenientes da camada de controle;

k) `GenerateCode`: classe responsável por preparar as informações configuradas pelo desenvolvedor e efetuar a chamada para a geração de código;

l) `VelocityController`: classe responsável por efetuar a geração de código;

m) `Convert`: classe responsável por converter `Strings` e tipos de dado do banco de dados em tipos equivalentes em Java.

O pacote `view` define as classes que compõem a camada de modelo. A figura 12 mostra o diagrama de classes do pacote `view`.

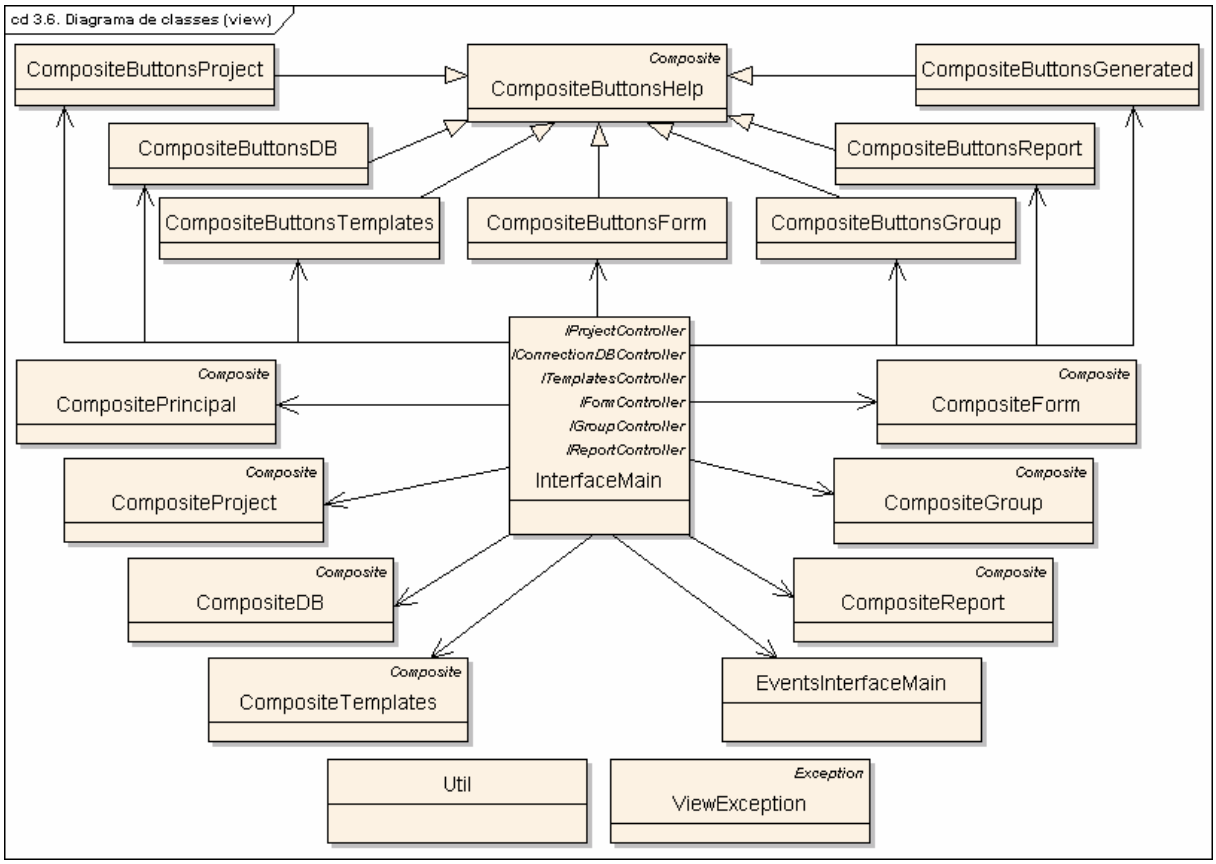


Figura 12 – Diagrama de classes do pacote `view`

Para cada classe que inicia com a nomenclatura `Composite`, existe uma outra classe com o mesmo nome, com prefixo `Events`. As classes `Events` foram criadas para que o código que implementa os eventos ficasse separado do código que implementa os componentes das

telas. As classes `Events` foram omitidas do diagrama da figura 12 para que se tenha uma melhor legibilidade da estrutura das classes. Segue o detalhamento das principais classes da camada de visão, relacionadas no diagrama da figura 12:

a) `InterfaceMain`: classe principal da camada de visão, efetuando a comunicação com a camada de controle. Ela é responsável por fazer a passagem das informações das telas para a classe `ProjectManager`, através da implementação das interfaces de controle, conforme definido na figura 11;
b) `EventsInterfaceMain`: classe que possui a implementação dos eventos da classe `InterfaceMain`;
c) `Util`: classe que efetua a formatação de mensagens ao usuário e apresenta as telas de seleção de arquivos e diretórios;
d) `ViewException`: classe responsável pelo tratamento de exceções provenientes da camada de visão.

3.3 ESPECIFICAÇÃO DA SAÍDA

Todo o código gerado é definido em *templates*, fazendo com que a saída não fique "amarrada" à ferramenta. Assim, é possível especificar a saída definindo os *templates* da forma desejada, podendo até ser utilizada uma outra linguagem de programação. O que deve ser respeitada é a estrutura de *templates* definida, que está fixa na implementação da ferramenta. A estrutura de *templates* adotada segue o padrão especificado pela MVC 2.

Cada formulário criado armazena uma referência para um objeto que representa uma tabela da base de dados. A partir dessa referência, são gerados arquivos para as camadas de modelo, controle e visão. Para a camada de modelo, são geradas classes Java que representam uma tabela do banco de dados, classes que implementam o padrão DAO e uma classe para efetuar a conexão com o banco de dados. Para a camada de controle são geradas classes Java que efetuam a execução das requisições, *servlets* que fazem as chamadas às classes de controle e o redirecionamento das páginas JSP e um arquivo de configuração que faz o registro dos *servlets*. Por fim, a camada de visão é gerada em arquivos JSP. As páginas geradas possibilitam o cadastro, alteração, exclusão e visualização das informações. Também é gerada uma página com um menu principal e opcionalmente podem ser geradas páginas que possibilitam a consulta de informações aplicando filtros. A figura 13 mostra a estrutura de

arquivos gerados pela ferramenta.

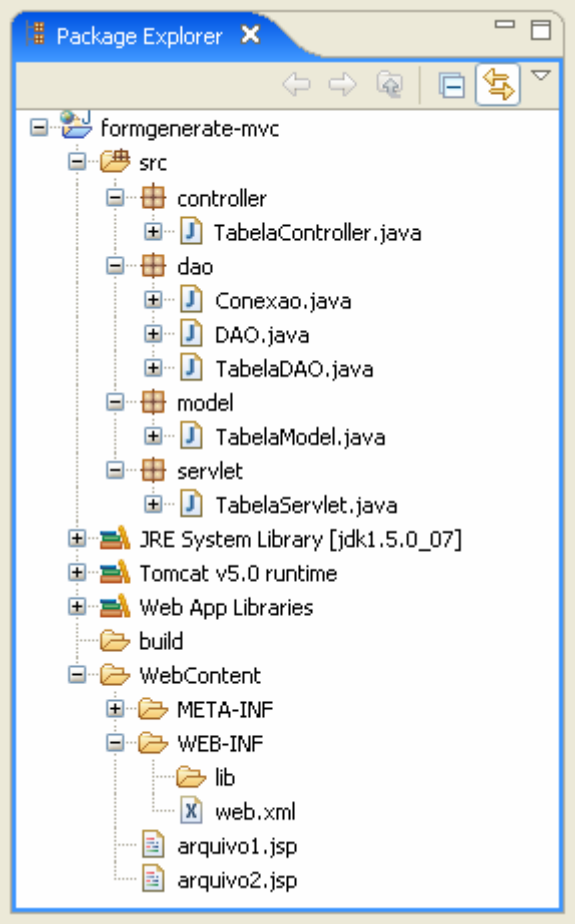


Figura 13 – Estrutura do código gerado

Na geração da camada de modelo, as classes que representam as tabelas do banco de dados seguem a nomenclatura `NomeTabelaModel.java`. É gerado um arquivo Java para cada tabela do banco de dados e cada campo da tabela é identificado como um atributo da classe. Os atributos possuem o mesmo nome definido na coluna da tabela, mas convertido para letras minúsculas. Para essas classes também são gerados construtores públicos (com e sem passagem de parâmetros) e métodos *getter* e *setter*. Para a implementação do padrão DAO são geradas classes Java que seguem o padrão de nomenclatura `NomeTabelaDAO.java` e são responsáveis pela comunicação com o banco de dados. Igualmente às classes de modelo, são gerados arquivos Java para cada tabela do banco de dados. Para estabelecer a conexão com o banco de dados é gerada uma classe chamada `Conexao.java`. Essa classe também é responsável por executar as instruções SQL provindas das classes que implementam o padrão DAO. O quadro 11 mostra a estrutura de código de uma classe, gerada pela ferramenta, que representa uma tabela do banco de dados.

```
public class CidadeModel {

   /*
    * Auto generate attributes
    */
   private long codigo_cidade;
   private String nome_cidade;
   private String sigla_estado;

   /*
    * Auto generate constructors
    */
   public CidadeModel() {
      super();
      this.codigo_cidade = 0;
      this.nome_cidade = "";
      this.sigla_estado = "";
   }

   public CidadeModel(long codigo_cidade,String nome_cidade,String sigla_estado){
      super();
      this.codigo_cidade = codigo_cidade;
      this.nome_cidade = nome_cidade;
      this.sigla_estado = sigla_estado;
   }

   /*
    * Auto generate methods getters and setters
    */
   public void setCodigo_cidade(long codigo_cidade) {
      this.codigo_cidade = codigo_cidade;
   }

   public long getCodigo_cidade() {
      return this.codigo_cidade;
   }

   public void setNome_cidade(String nome_cidade) {
      this.nome_cidade = nome_cidade;
   }

   public String getNome_cidade() {
      return this.nome_cidade;
   }

   public void setSigla_estado(String sigla_estado) {
      this.sigla_estado = sigla_estado;
   }

   public String getSigla_estado() {
      return this.sigla_estado;
   }

}
```

Quadro 11 – Estrutura de uma classe de modelo gerada pela ferramenta

O quadro 12 mostra a estrutura de código da classe, gerada pela ferramenta, que efetua a conexão com o banco de dados.

```
public class Conexao {

     private String command;
     private static Connection conn;
     private static Conexao conexao;
     private Statement stmt;

     public Conexao() throws InstantiationException, IllegalAccessException,
                             ClassNotFoundException, SQLException {
          String db = "NOME_BANCO";
          String server = "SERVIDOR";
          String database = "NOME_BASE_DE_DADOS";
          String user = "NOME_USUARIO";
          String password = "SENHA_USUARIO";
          if ( password == null ) { password = "''"; }
             Class.forName( getDriver( db ) ).newInstance();
             String url = mountURL( db, server, database );
             conn = DriverManager.getConnection( url, user, password );
     }
     private static String mountURL(String db, String server, String database) {
          if ( db.equals( "ORACLE" ) ) {
             return "jdbc:oracle:thin:@" + server + ":1521:" + database;
          }
          if ( db.equals( "SQLSERVER" ) ) {
             return "jdbc:odbc:" + server;
          }
          return "jdbc:mysql://" + server + ":3306/" + database;
     }
     private static String getDriver( String db ) {
          if ( db.equalsIgnoreCase( "ORACLE" ) ) {
             return "oracle.jdbc.driver.OracleDriver";
          }
          if ( db.equalsIgnoreCase( "SQLSERVER" ) ) {
             return "sun.jdbc.odbc.JdbcOdbcDriver";
          }
          return "com.mysql.jdbc.Driver";
     }
     public ResultSet openSQL( PreparedStatement pstmt ) throws SQLException {
        return pstmt.executeQuery();
     }
     public PreparedStatement setPreparedStatement( String sql ) throws
                                                                 SQLException {
        return conn.prepareStatement( sql );
     }
     public void execSQL( PreparedStatement pstmt ) throws SQLException {
        try {
           pstmt.execute();
        } finally {
           pstmt.close();
        }
     }
     public static Conexao getInstanceOf() {
        if ( conexao == null ) {
           try {
              conexao = new Conexao();
              return conexao;
           } catch (Exception e) { return null; }
        }
        else return conexao;
     }
     ...
}
```

Quadro 12 – Estrutura da classe de conexão com o banco de dados gerada pela ferramenta

O quadro 13 mostra a estrutura de código de uma classe DAO gerada pela ferramenta.

```
public class PaisDAO {

   private Conexao conexao;
   private PaisModel paisModel;

   public PaisDAO() throws Exception { this.conexao = Conexao.getInstanceOf(); }

   public void setPaisModel( PaisModel paisModel ) {
      this.paisModel = paisModel;
   }
   public void insert() throws SQLException {
      if ( this.paisModel != null ) {
         PreparedStatement pstmt = this.conexao.
                        setPreparedStatement( "INSERT INTO PAIS
                        (CODIGO_PAIS,NOME_PAIS) values (?,?)" );
         pstmt.setLong( 1, paisModel.getCodigo_pais() );
         pstmt.setString( 2, paisModel.getNome_pais() );
         this.conexao.execSQL( pstmt );
      }
   }
   public void update() throws SQLException {
      if ( this.paisModel != null ) {
         PreparedStatement pstmt = this.conexao.setPreparedStatement( "UPDATE
               PAIS SET CODIGO_PAIS = ?, NOME_PAIS = ? WHERE CODIGO_PAIS = ?" );
         pstmt.setLong( 1, paisModel.getCodigo_pais() );
         pstmt.setString( 2, paisModel.getNome_pais() );
         pstmt.setLong( 3, paisModel.getCodigo_pais() );
         this.conexao.execSQL( pstmt );      }
   }
   public void delete() throws SQLException {
      if ( this.paisModel != null ) {
         PreparedStatement pstmt = this.conexao.setPreparedStatement( "DELETE
                                   PAIS WHERE CODIGO_PAIS = ?" );
         pstmt.setLong( 1, paisModel.getCodigo_pais() );
         this.conexao.execSQL( pstmt );
      }
   }
   public PaisModel getPaisModel( long codigo_pais ) throws SQLException {
      PaisModel paisModel = null;
      PreparedStatement pstmt = this.conexao.setPreparedStatement( "SELECT * FROM
                                PAIS WHERE CODIGO_PAIS = ?" );
      pstmt.setLong( 1, codigo_pais );
      ResultSet select = this.conexao.openSQL( pstmt );
      while ( select.next() ) {
         paisModel = new PaisModel();
         paisModel.setCodigo_pais( select.getLong( "CODIGO_PAIS" ) );
         paisModel.setNome_pais( select.getString( "NOME_PAIS" ) );
      }
      select.close();
      pstmt.close();
      return paisModel;
   }
   public List select() throws SQLException {
      List<PaisModel> objects = new ArrayList<PaisModel>();
      PaisModel paisModel = null;
      PreparedStatement pstmt = this.conexao.setPreparedStatement( "SELECT * FROM
                                PAIS ORDER BY 1" );
      ResultSet select = this.conexao.openSQL( pstmt );
      while ( select.next() ) {
         paisModel = this.carregaDados( select );
         objects.add( paisModel );
      }
      select.close();
      return objects;
   }
```

```
   public List select( List types, List params, List values, List operators )
                                                          throws SQLException {
      List<PaisModel> objects = new ArrayList<PaisModel>();
      PaisModel paisModel = null;
      String sql = "SELECT * FROM PAIS WHERE ";
      for ( int i = 0; i < params.size(); i++ ) {
         if ( i == params.size() - 1 )
            sql+=params.get(i)+getOperatorSelect((String)operators.get(i));
         sql+=params.get(i)+getOperatorSelect((String)operators.get(i))+" AND ";
      }
      PreparedStatement pstmt = this.conexao.setPreparedStatement( sql );
      for ( int i = 0; i < values.size(); i++ ) {
         if ( types.get(i).equals("String") || types.get(i).equals("char") )
            pstmt.setString( i + 1, (String) values.get(i) );
         if ( types.get(i).equals("long") )
            pstmt.setLong( i + 1, ((Long) values.get(i)).longValue() );
         if ( types.get(i).equals( "double" ) )
            pstmt.setDouble( i + 1, ((Double) values.get(i)).doubleValue() );
         if ( types.get( i ).equals( "java.sql.Date" ) )
            pstmt.setDate( i + 1, (java.sql.Date) values.get( i ) );
      }
      ResultSet select = this.conexao.openSQL( pstmt );
      while ( select.next() ) {
         paisModel = this.carregaDados( select );
         objects.add( paisModel );
      }
      select.close();
      pstmt.close();
      return objects;
   }
   private PaisModel carregaDados( ResultSet select ) throws SQLException {
      PaisModel paisModel = new PaisModel();
      paisModel.setCodigo_pais( select.getLong( "CODIGO_PAIS" ) );
      paisModel.setNome_pais( select.getString( "NOME_PAIS" ) );
      return paisModel;
   }
   private String getOperatorSelect( String operator ) {
      if ( operator.equalsIgnoreCase( "Igual" ) ) return " = ?";
      if ( operator.equalsIgnoreCase( "Diferente" ) ) return " <> ?";
      if ( operator.equalsIgnoreCase( "Maior" ) ) return " > ?";
      if ( operator.equalsIgnoreCase( "MaiorIgual" ) ) return " >= ?";
      if ( operator.equalsIgnoreCase( "Menor" ) ) return " < ?";
      if ( operator.equalsIgnoreCase( "MenorIgual" ) ) return " <= ?";
      if ( operator.equalsIgnoreCase( "Contendo" ) ) return " LIKE '%?%'";
      if ( operator.equalsIgnoreCase( "Iniciando" ) ) return " LIKE '%?'";
      if ( operator.equalsIgnoreCase( "Terminando" ) ) return " LIKE '?%'";
      return null;
   }

}
```

Quadro 13 – Estrutura de uma classe DAO gerada pela ferramenta

Para a camada de controle são geradas classes Java de controle, responsáveis por instanciar os objetos das classes de modelo e efetuar as chamadas às classes DAO. Para cada tabela do banco de dados é gerada uma classe de controle e as mesmas seguem a nomenclatura `NomeTabelaController.java`. Para essa camada também são gerados *servlets* que fazem o controle das requisições vindas da camada de visão. Os *servlets* recebem as informações digitadas nas páginas JSP e, de acordo com a ação passada como parâmetro, encaminham as solicitações para as classes de controle. Os *servlets* também são responsáveis

por efetuar o redirecionamento das páginas JSP e invocar a execução dos relatórios. Eles são gerados seguindo a nomenclatura `NomeTabelaServlet.java` e `NomeTabelaServletReport.java`. É gerado também um arquivo chamado `web.xml`. Esse arquivo contém o registro de todos os *servlets* gerados automaticamente pela ferramenta. Os quadros 14 a 16 apresentam as estruturas de código das classes de controle, dos *servlets* e do arquivo `web.xml`, todos gerados automaticamente pela ferramenta.

```
public class EstadoController {

   private EstadoDAO estadoDAO = null;

   public EstadoController( EstadoDAO estadoDAO ) {
      this.estadoDAO = estadoDAO;
   }

   public void inserir(String sigla_estado, String nome_estado, long codigo_pais)
                    throws SQLException {
      this.estadoDAO.insert( this.newEstado( sigla_estado, nome_estado,
                             codigo_pais ) );
   }

   public void alterar(String sigla_estado, String nome_estado, long codigo_pais)
                    throws SQLException {
      this.estadoDAO.update( this.newEstado( sigla_estado, nome_estado,
                             codigo_pais ) );
   }

   public void excluir( String sigla_estado ) throws SQLException {
      EstadoModel estadoModel = new EstadoModel();
      estadoModel.setSigla_estado( sigla_estado );
      this.estadoDAO.delete( estadoModel );
   }

   public List getObjects() throws SQLException {
      return this.estadoDAO.select();
   }

   public List getObjects(List types, List params, List values, List operators)
                    throws SQLException {
      return this.estadoDAO.select( types, params, values, operators );
   }

   public EstadoDAO getEstadoDAO() {
      return this.estadoDAO;
   }

   private EstadoModel newEstado( String sigla_estado, String nome_estado,
                                  long codigo_pais ) {
      EstadoModel estadoModel = new EstadoModel();
      estadoModel.setSigla_estado( sigla_estado );
      estadoModel.setNome_estado( nome_estado );
      estadoModel.setCodigo_pais( codigo_pais );
      return estadoModel;
   }

}
```

Quadro 14 – Estrutura de uma classe de controle gerada pela ferramenta

```
public class PaisServlet extends HttpServlet {

   private static final long serialVersionUID = 1L;
   private PaisController paisController = null;

   public void init() throws ServletException {
      super.init();
      try {
         this.paisController = new PaisController( new PaisDAO() );
      } catch (Exception e) { System.out.println( e.getMessage() ); }
   }
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {
      this.doPost( request, response );
   }
   protected void doPost(HttpServletRequest request,HttpServletResponse response)
                        throws ServletException, IOException {
      String action = request.getParameter( "action" );
      if ( action != null ) { this.paisRegister( request, response ); }
      else {
         RequestDispatcher reqDisp = getServletContext().getRequestDispatcher(
                                 "/erro.jsp?erro=Página não encontrada!" );
         reqDisp.forward( request, response );
      }
   }
   private void paisRegister( HttpServletRequest request, HttpServletResponse
                              response ) throws ServletException, IOException {
      try {
         String urlDestino = "";
         Util util = new Util();
         String action = request.getParameter( "action" );
         if ( action != null ) {
            if ( action.equalsIgnoreCase( "pesquisar" ) ) {
               urlDestino = "/PaisSearch.jsp?colorGrid1=" + util.getCorGrid( 0 )
                            + "&colorGrid2=" + util.getCorGrid( 1 );
            }
            else {
               long codigo_pais = Long.parseLong( util.formatToParseLong(
                                  request.getParameter( "codigo_pais" ) ) );
               String nome_pais = request.getParameter( "nome_pais" );
               if ( action.equalsIgnoreCase( "actionIns" ) ) {
                  this.paisController.inserir( codigo_pais, nome_pais );
               }
               if ( action.equalsIgnoreCase( "actionUpd" ) ) {
                  this.paisController.alterar( codigo_pais, nome_pais );
               }
               if ( action.equalsIgnoreCase( "actionDel" ) ) {
                  this.paisController.excluir( codigo_pais );
               }
               urlDestino = "/PaisIndex.jsp?colorGrid1=" + util.getCorGrid( 0 )
                            + "&colorGrid2=" + util.getCorGrid( 1 );
            }
            List objects = this.paisController.getObjects();
            request.setAttribute( "objects", objects );
            RequestDispatcher reqDisp = getServletContext().getRequestDispatcher(
                                        urlDestino );
            reqDisp.forward( request, response );
         }
      }
      catch (SQLException e) {
           response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
      }
   }
}
```

Quadro 15 – Estrutura de um *servlet* gerado pela ferramenta

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
       <display-name>
       T1 - ControlePassagens</display-name>
       <welcome-file-list>
              <welcome-file>index.html</welcome-file>
              <welcome-file>index.htm</welcome-file>
              <welcome-file>index.jsp</welcome-file>
              <welcome-file>default.html</welcome-file>
              <welcome-file>default.htm</welcome-file>
              <welcome-file>default.jsp</welcome-file>
       </welcome-file-list>

       <servlet>
          <servlet-name>PaisServlet</servlet-name>
          <servlet-class>servlet.PaisServlet</servlet-class>
       </servlet>

       <servlet-mapping>
          <servlet-name>PaisServlet</servlet-name>
          <url-pattern>/pais</url-pattern>
       </servlet-mapping>

       <servlet>
          <servlet-name>EstadoServlet</servlet-name>
          <servlet-class>servlet.EstadoServlet</servlet-class>
       </servlet>

       <servlet-mapping>
          <servlet-name>EstadoServlet</servlet-name>
          <url-pattern>/estado</url-pattern>
       </servlet-mapping>

      ...

</web-app>
```

Quadro 16 – Estrutura do arquivo de registro dos *servlets* gerado pela ferramenta

A camada de visão é gerada em arquivos JSP. De acordo com a definição dos *templates*, as páginas geradas possibilitam ao usuário: listar as informações obtidas do banco de dados; efetuar o cadastro e alteração das informações; exibir páginas de pesquisa; e efetuar a execução de relatórios. As páginas JSP são geradas com nomenclaturas de acordo com suas funcionalidades. As páginas que listam as informações do banco de dados são geradas com a nomenclatura `NomeTabelaIndex.jsp`, as páginas para o cadastro e alterações das informações são geradas como `NomeTabelaRegister.jsp` e as páginas de pesquisa seguem o padrão `NomeTabelaSearch.jsp`. São gerados também arquivos com a nomenclatura `NomeTabelaExecReport.jsp` para a execução dos relatórios, e `NomeTabelaShowReport.jsp` para a visualização das informações obtidas após a execução dos relatórios. Também é gerado um arquivo chamado `Menu.jsp`, por onde é feita a chamada das páginas de visualização das informações e dos relatórios. Os quadros 17 a 21 mostram a estruturação das páginas JSP geradas pela ferramenta.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<link rel="stylesheet" type="text/css" href="folhaEstilo.css">
<script
  src="scripts.js">
</script>
<html>
<body>
<table width="100%">
  <tr align="right">
    <td align="right">
      <a href="pais?action=actionIns"
         target="main"><img src="images/novo.gif" border="0">Incluir
      </a>
    </td>
  </tr>
</table>
<table class="tableForm" width="100%">
  <tr>
    <td class="tituloTelas" colspan="8"><b> :: PAIS : </b></td>
  </tr>
  <tr>
    <td class="tituloTabela" width="1%"></td>
    <td align="left" class="tituloTabela"><b>Código</b></td>
    <td align="left" class="tituloTabela"><b>País</b></td>
    <td class="tituloTabela" width="1%"></td>
  </tr>
  <jstlTag:forEach items="${objects}" var="objects" varStatus="linha">
    <jstlTag:if test="${linha.index % 2 == 0}">
      <jstlTag:set var="cor" value="${param.colorGrid1}"/>
    </jstlTag:if>
    <jstlTag:if test="${linha.index % 2 != 0}">
      <jstlTag:set var="cor" value="${param.colorGrid2}"/>
    </jstlTag:if>
    <tr bgcolor="${cor}">
      <td align="center">
        <a href="pais?action=actionUpd&codigo_pais=${objects.codigo_pais}"
           target="main">
          <img src="images/form.gif" border="0" alt="Editar Registro">
        </a>
      </td>
      <td align="left"><jstlTag:out value="${objects.codigo_pais}"/></td>
      <td align="left"><jstlTag:out value="${objects.nome_pais}"/></td>
      <td align="center">
        <a href="pais?action=actionDel&codigo_pais=${objects.codigo_pais}"
           target="main">
          <img src="images/deleteGrid.gif" border="0"alt="Excluir Registro">
        </a>
      </td>
    </tr>
  </jstlTag:forEach>
</table>
<br>
<b>Legenda</b>
<table border="0">
  <tr align="center">
    <td><img src="images/form.gif" border="0">Editar Registro</td>
  </tr>
  <tr align="center">
    <td><img src="images/deleteGrid.gif" border="0">Excluir Registro</td>
  </tr>
</table>
</body>
</html>
<html>
```

Quadro 17 – Página JSP que lista as informações de uma tabela do banco de dados

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<link rel="stylesheet" type="text/css" href="folhaEstilo.css">
<script
  src="scripts.js">
</script>
</head>

<jstlTag:set var="codigo_pais" value="${param.codigo_pais}"/>
<jstlTag:set var="nome_pais" value="${param.nome_pais}"/>
<jstlTag:set var="action" value="${param.action}"/>

<jstlTag:if test="${action == 'actionIns'}">
  <jstlTag:set var="action" value="inserir"/>
  <jstlTag:set var="status" value="Inserindo Registro"/>
</jstlTag:if>

<jstlTag:if test="${action == 'actionUpd'}">
  <jstlTag:set var="action" value="alterar"/>
  <jstlTag:set var="status" value="Editando Registro"/>
</jstlTag:if>

<body onload="initFocus()">
<form action="pais" method="post" name="pais">
<table class="tableForm" align="center">
  <tr>
    <td class="tituloTelas" colspan="6"><b>:: PAIS ::</b></td>
  </tr>
  <tr>
    <td class="tituloTabela" colspan=6><b>${status}</b></td>
  </tr>
  <tr>
    <td align="right" classe="tdLabelCadastros">Código</td>
    <td align="left" class="tdInputCadastros"><input
        class="inputTextCodigo2" type="text" name="codigo_pais"
        value="${codigo_pais}" onkeydown="mascaraInt(this,event)"/>
    </td>
  </tr>
  <tr>
    <td align="right" classe="tdLabelCadastros">País</td>
    <td align="left" class="tdInputCadastros"><input
        class="inputText" type="text" name="nome_pais" value="${nome_pais}"/>
    </td>
  </tr>
  <tr>
  </tr>
  <tr>
    <td align="right" colspan="6"><input class="inputButton"
       type="submit" value="Confirmar"></td>
  </tr>
</table>
  <input type="hidden" name="action" value="${action}"></form>
</body>
</html>
```

Quadro 18 – Página JSP para o cadastro e alteração das informações de uma tabela do banco de dados

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<link rel="stylesheet" type="text/css" href="folhaEstilo.css">
<script src="scripts.js"></script>
<script language="JavaScript">
  function returnForm(key,value) {
    top.window.opener.document.locacao.codigo_cliente.value=key;
    top.window.opener.document.locacao.nome_cliente.value=value;
    top.window.opener.document.locacao.codigo_cliente.focus();
    window.close();
  }
</script>
</head>
<body>
<table class="tableForm" width="100%">
  <tr>
    <td class="tituloTelas" colspan="8"><b>:: PESQUISA LOCACAO ::</b></td>
  </tr>
  <tr>
    <td align="left" class="tituloTabela"><b>CODIGO_CLIENTE</b></td>
    <td align="left" class="tituloTabela"><b>NOME_CLIENTE</b></td>
    <td align="left" class="tituloTabela"><b>RG_CLIENTE</b></td>
    <td align="left" class="tituloTabela"><b>CPF_CLIENTE</b></td>
    <td align="left" class="tituloTabela"><b>FONE_CLIENTE</b></td>
    <td align="left" class="tituloTabela"><b>CODIGO_ENDERECO</b></td>
    <td align="left" class="tituloTabela"><b>CONTATO_CLIENTE</b></td>
    <td align="left" class="tituloTabela"><b>FONE_CONTATO</b></td>
  </tr>
  <jstlTag:forEach items="${objects}" var="objects" varStatus="linha">

    <jstlTag:if test="${linha.index % 2 == 0}">
      <jstlTag:set var="cor" value="${param.colorGrid1}"/>
    </jstlTag:if>

    <jstlTag:if test="${linha.index % 2 != 0}">
      <jstlTag:set var="cor" value="${param.colorGrid2}"/>
    </jstlTag:if>

    <tr bgcolor="${cor}">
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.codigo_cliente}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.nome_cliente}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.rg_cliente}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.cpf_cliente}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.fone_cliente}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.codigo_endereco}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.contato_cliente}</a></td>
      <td align="left"><a href="JavaScript:returnForm(${objects.codigo_cliente},
         '${objects.nome_cliente}');">${objects.fone_contato}</a></td>
    </tr>
  </jstlTag:forEach>
</table>
</body>
</html>
```

Quadro 19 – Página JSP que exibe uma tela de pesquisa

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="folhaEstilo.css">
<script src="scripts.js"></script>
</head>
<body>
<form action="locacao_report?action=pesquisar" method="post" name="locacao">
<table class="tableForm" align="left">
  <tr><td class="tituloTelas" colspan="6"><b>:: LOCACAO ::</b></td></tr>
  <tr>
    <td align="right" classe="tdLabelCadastros">Cliente</td>
    <td class="tdInputCadastros">
      <select name="codigo_cliente_option">
        <option value="Maior">Maior que</option>
        <option value="MaiorIgual">Maior ou igual à</option>
        <option value="Menor">Menor que</option>
        <option value="MenorIgual">Menor ou igual à</option>
        <option value="Igual">Igual à</option>
        <option value="Diferente">Diferente de</option>
      </select>
    </td>
    <td align="left" class="tdInputCadastros">
      <input class="inputText" type="text" name="codigo_cliente_input" value=""
             onkeydown="mascaraInt(this,event)"/></tr>
  <tr>
    <td align="right" classe="tdLabelCadastros">Data da locação</td>
    <td class="tdInputCadastros">
      <select name="data_locacao_option">
        <option value="Maior">Maior que</option>
        <option value="MaiorIgual">Maior ou igual à</option>
        <option value="Menor">Menor que</option>
        <option value="MenorIgual">Menor ou igual à</option>
        <option value="Igual">Igual à</option>
        <option value="Diferente">Diferente de</option>
      </select></td>
    <td align="left" class="tdInputCadastros">
      <input class="inputTextData" type="text" name="data_locacao_input" value=""
             onkeydown="mascaraData(this,event)"/></tr>
    <tr>
      <td align="right" classe="tdLabelCadastros">Data de devolução</td>
      <td class="tdInputCadastros">
        <select name="data_devolucao_option">
          <option value="Maior">Maior que</option>
          <option value="MaiorIgual">Maior ou igual à</option>
          <option value="Menor">Menor que</option>
          <option value="MenorIgual">Menor ou igual à</option>
          <option value="Igual">Igual à</option>
          <option value="Diferente">Diferente de</option>
        </select>
      </td>
      <td align="left" class="tdInputCadastros">
        <input class="inputTextData" type="text" name="data_devolucao_input"
               value="" onkeydown="mascaraData(this,event)"/>
    </tr>
    <tr>
      <td align="right" colspan="6"><input class="inputButton"
          type="submit" value="Executar">
      </td>
    </tr>
</table>
</form>
</body>
</html>
```

Quadro 20 – Página JSP que executa um relatório

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<link rel="stylesheet" type="text/css" href="folhaEstilo.css">
<script src="scripts.js"></script>
</head>

<body>
<table class="tableForm" width="100%">
  <tr>
    <td class="tituloTelas" colspan="10"><b>:: LOCACAO ::</b></td>
  </tr>
  <tr>
    <td align="left" class="tituloTabela"><b>Locação</b></td>
    <td align="left" class="tituloTabela"><b>Data da locação</b></td>
    <td align="left" class="tituloTabela"><b>Data de devolução</b></td>
    <td align="left" class="tituloTabela"><b>Total a pagar</b></td>
  </tr>

  <jstlTag:forEach items="${objects}" var="objects" varStatus="linha">
    <jstlTag:if test="${linha.index % 2 == 0}">
      <jstlTag:set var="cor" value="${param.colorGrid1}"/>
    </jstlTag:if>

    <jstlTag:if test="${linha.index % 2 != 0}">
      <jstlTag:set var="cor" value="${param.colorGrid2}"/>
    </jstlTag:if>

    <tr bgcolor="${cor}">
      <td align="left"><jstlTag:out value="${objects.codigo_locacao}"/></td>
      <td align="left"><jstlTag:out value="${objects.data_locacao}"/></td>
      <td align="left"><jstlTag:out value="${objects.data_devolucao}"/></td>
      <td align="left"><jstlTag:out value="${objects.valor_total}"/></td>
    </tr>
  </jstlTag:forEach>

</table>
</body>
</html>
```

Quadro 21 – Página JSP que exibe o retorno da execução de um relatório

3.4 IMPLEMENTAÇÃO

A implementação de FormGenerate é fortemente baseada na ferramenta CodGer, desenvolvida por Menin (2005). Mesmo possuindo similaridades, FormGenerate foi desenvolvido de forma diferente de CodGer, utilizando outra estrutura de aplicação e outros recursos para a extração do metadados e geração de código. Nessa seção são descritas as ferramentas e técnicas utilizadas no desenvolvimento da ferramenta e é demonstrada a sua operacionalidade.

3.4.1 Técnicas e ferramentas utilizadas

A ferramenta aqui apresentada foi desenvolvida na linguagem Java, utilizando o ambiente de desenvolvimento Eclipse 3.2, a biblioteca gráfica SWT, JDK versão 1.5.0 e adotando o padrão MVC. Para a geração do código de saída foi utilizado o motor de *templates* Velocity, através da biblioteca `velocity-dep-1.4.jar`. Para efetuar a leitura do metadados foi utilizada a API JDBC, através das bibliotecas `ojdbc14.jar` e `mysql-connector-java-3.1.13-bin.jar` e dos *drivers* JDBC `oracle.jdbc.driver.OracleDriver`, `com.mysql.jdbc.Driver` e `sun.jdbc.odbc.JdbcOdbcDriver`. Para os testes de geração de código foi especifica uma modelagem para bancos de dados relacionais, representando uma locadora de filmes, a qual foi criada nos bancos de dados Oracle XE, Microsoft SQL Server 2000 e MySQL 5.0.21.

3.4.2 Implementação da ferramenta

Nessa seção é apresentado o desenvolvimento da ferramenta, mostrando e explicando alguns trechos de código fonte. A seção está divida em: implementação da camada de modelo, implementação da camada de controle e implementação da camada de visão.

3.4.2.1 Implementação da camada de modelo

A camada de modelo está definida no pacote `model` e este, por sua vez, está subdividido em `model.metadata` e `model.project`. No pacote `model.metadata` estão as classes `SgbdModel`, `ConstraintModel`, `PrimaryKeyModel`, `ForeignKeyModel`, `TableModel` e `ColumnModel`, que armazenam as informações extraídas do metadados, efetuando assim o mapeamento dos objetos do banco para uma estrutura em memória. Nesse pacote também há a classe `LoadDatabase`, responsável por extrair as informações do metadados. O quadro 22 apresenta a implementação de um método da classe `LoadDatabase`. Esse método lê as colunas de uma tabela da base de dados, recuperando suas informações para memória.

```
package model.metadata;

public class LoadDatabase {

   ...

   public static List<ColumnModel> loadColumns(DatabaseMetaData metadata,
                                    TableModel table) throws SQLException {

      // retorna um ResultSet contendo as colunas da tabela passada como
         parâmetro
      ResultSet rs = metadata.getColumns(null, null, table.getName(), null);
      List<ColumnModel> columns = new ArrayList<ColumnModel>();

      while ( rs.next() ) {

         // obtém as informações das colunas
         boolean nul = false;
         String name = rs.getString( "COLUMN_NAME" );
         String type = rs.getString( "TYPE_NAME" );
         int size = rs.getInt( "COLUMN_SIZE" );
         int precision = rs.getInt( "DECIMAL_DIGITS" );
         int seq = rs.getInt( "ORDINAL_POSITION" );
         String allowNull = rs.getString( "IS_NULLABLE" );

         if ( allowNull.equalsIgnoreCase( "YES" ) ) {
            nul = true;
         }

         // cria um objeto de ColumnModel
         ColumnModel column = new ColumnModel(seq, name, type, size,
                                              precision, nul, null);

         // adiciona o objeto de ColumnModel nas lista de colunas
         columns.add( column );

      }

      // retorna a lista de colunas da tabela
      return columns;
   }

   ...

}
```

Quadro 22 – Método que lê as informações das colunas de uma tabela da base de dados

A classe `SQLCommands` é responsável por efetuar a conexão e a desconexão com o banco de dados. Ela utiliza os atributos `static final` definidos nas interfaces `IDB` e `IDriver`. Os atributos definidos nas interfaces correspondem ao nome dos bancos de dados utilizados e o nome dos seus respectivos *drivers* JDBC. O quadro 23 apresenta o código fonte do método que efetua a conexão com o banco de dados.

```
package model.metadata;

public class SQLCommands {

   ...

   public static Connection connect( String db, String server, String
                       database, String user, String password ) throws
                       InstantiationException, IllegalAccessException,
                       ClassNotFoundException, SQLException {

      // seleciona o driver JDBC de acordo com o banco de dados
      String driver = selectDriver( db );
      if ( password == null ) {
         password = "''";
      }

      // registra o driver JDBC na JVM
      Class.forName( driver ).newInstance();

      // monta a URL de acordo com o banco de dados
      String url = mountURL( db, server, database );

      // retorna a conexão com o banco de dados
      return DriverManager.getConnection( url, user, password );
   }

   ...

}
```

Quadro 23 – Método que efetua a conexão com o banco de dados

O pacote `model.project` é composto pelas classes `ProjectModel`, `ConnectionDBModel`, `TemplateModel`, `DirectoryModel`, `FormModel`, `ColumnFormModel`, `GroupModel`, `ReportModel`, `ColumnReportModel` e pelas interfaces `IPurposeDirectories`, `IPurposeTemplates`. Os objetos das classes desse pacote armazenam as informações configuradas pelo desenvolvedor durante o uso da ferramenta, e essas informações necessitam ser salvas e posteriormente recuperadas para a memória. Para atender a essa necessidade, as classes do pacote `model.project`, assim como as classes que efetuam o mapeamento do banco de dados, foram definidas seguindo o padrão JavaBean, onde é necessário existir um construtor público sem passagem de parâmetros e, para cada atributo, devem ser criados os seus respectivos métodos *getter* e *setter*. O quadro 24 apresenta o código fonte de uma classe que segue o padrão JavaBean.

As interfaces `IPurposeDirectories` e `IPurposeTemplates` possuem atributos `static final` que indicam a finalidade de cada *template* e cada diretório do projeto. A camada de controle utiliza os atributos dessas interfaces para criar e recuperar os diretórios e *templates* do projeto.

```
package model.project;

public class ConnectionDBModel implements Serializable {

   private String db;
   private String server;
   private String database;
   private String user;
   private String password;
   private static final long serialVersionUID = 1L;

   /** Construtor sem passagem de parâmetros */

   public ConnectionDBModel() {
      super();
      this.db = null;
       this.server = null;
       this.database = null;
       this.user = null;
       this.password = null;
   }

   /** Métodos getters e setters*/

   public String getDb() {
      return db;
   }

   public String getServer() {
      return server;
   }

   public String getDatabase() {
      return database;
   }

   public String getUser() {
      return user;
   }

   public String getPassword() {
      return password;
   }

   public void setDb( String db ) {
      this.db = db;
   }

   public void setServer( String server ) {
      this.server = server;
   }

   public void setDatabase( String database ) {
      this.database = database;
   }

   public void setUser( String user ) {
      this.user = user;
   }

   public void setPassword( String password ) {
      this.password = password;
   }

}
```

Quadro 24 – Classe definida no padrão JavaBean

3.4.2.2 Implementação da camada de controle

A camada de controle está definida no pacote `controller` e este, por sua vez, está subdividido em `controller.commons`, `controller.generate` e `controller.project`.

Em `controller.commons` estão definidas algumas classes de uso comum na camada de controle. Essas classes são responsáveis por executar funcionalidades auxiliares como: efetuar a leitura e gravação de arquivos e, gravar e recuperar objetos serializados.

No pacote `controller.project` estão definidas as interfaces que especificam os serviços que a camada de controle disponibiliza. Essas interfaces são implementadas pela camada de visão possibilitando assim a comunicação entre as duas camadas. Esse pacote também possui a classe `ProjectManager`, que é responsável por executar todas as requisições vindas da camada de visão e atualizar as informações da camada de modelo.

Em `controller.generate` estão as classes responsáveis por preparar e efetuar a geração de código. Esse pacote é composto pelas classes `Convert`, `GenerateCode` e `VelocityController`. A classe `GenerateCode` é responsável por preparar as informações configuradas pelo desenvolvedor na ferramenta e fazer a chamada ao Velocity passando como parâmetros: o diretório onde estão armazenados os *templates*, o nome do *template* que será utilizado no *merge*, os objetos que possuem as informações que serão substituídas no *template* no momento do *merge* e o nome do arquivo a ser gerado. A classe `VelocityController` recebe essas informações e prepara o Velocity para iniciar a geração de código. Os procedimentos mais básicos, e que foram utilizados para que o Velocity efetue a geração de código foram os seguintes:

a) instanciar a *engine*: faz com que o ambiente de execução do Velocity seja carregado para a memória e todas as suas funcionalidades fiquem disponíveis;
b) criar contexto que será utilizado pelo Velocity: é o mapeamento dos objetos Java que serão utilizados dentro do *template*. Ou seja, é a nomeação dos objetos em forma textual, para que os mesmos sejam referenciados dentro do *template* através da VTL. Dessa forma, é possível acessar os métodos dos objetos no momento em que o Velocity efetua a análise do *template*;
c) inicializar a *engine*: é passado como parâmetro o caminho que identifica o local onde o Velocity irá buscar o arquivo *template*. Caso essa informação não seja utilizada na inicialização da *engine*, o Velocity utilizará o caminho padrão, que é a raiz do projeto onde a aplicação foi instanciada;

d) obter o conteúdo do *template*: é feito através da *engine*, e é armazenado em um objeto do tipo `Template`, próprio do Velocity. O objeto `Template` armazena o conteúdo do *template* em memória, na forma de uma estrutura de dados conhecida como AST;

e) realizar o *merge* das informações: é a substituição das referências definidas no *template*, através da linguagem VTL, pelos valores armazenados nos atributos dos objetos Java presentes no contexto passado ao Velocity.

O resultado obtido após o *merge* é disponibilizado em um objeto do tipo `StringWriter`. Esse objeto pode ser manipulado pelo desenvolvedor conforme a sua necessidade. Na implementação do FormGenerate, o resultado do *merge* é gravado em um arquivo de extensão `.java`, `.jsp` ou `.xml`. O quadro 25 apresenta o código da classe `VelocityController`.

```
package controller.generate;

public class VelocityController {

   public static void generate( String packageTemplates,
               String nameTemplate, Map context, String fileName ) throws
                      ResourceNotFoundException, ParseErrorException,
                      MethodInvocationException, IOException, Exception {

      // instancia os objetos necessários
      VelocityEngine velociteEngine = new VelocityEngine();
      VelocityContext velociteContext = new VelocityContext();
      Properties pathTemplates = new Properties();

      // seta o properties com o path onde encontram-se os templates
      pathTemplates.put( "file.resource.loader.path", packageTemplates );

      // inicia a engine do Velocity
      velociteEngine.init( pathTemplates );

      // obtém o template
      Template template = velociteEngine.getTemplate( nameTemplate );

      // seta o contexto dos objetos utilizados no merge
      for ( Iterator it = context.keySet().iterator(); it.hasNext(); ) {
         String key = (String) it.next();
         velociteContext.put( key, context.get( key ) );
      }

      // faz o merge
      StringWriter writer = new StringWriter();
      template.merge( velociteContext, writer );

      // grava o arquivo em disco
      Util.writeFile( fileName, writer.toString() );
   }

}
```

Quadro 25 – Classe que efetua a geração de código

3.4.2.3 Implementação da camada de visão

A camada de visão está definida no pacote `view` e este, por sua vez, está subdividido em `view.commons`, `view.compositesButtons`, `view.compositesWork` e `view.principal`. A divisão foi definida dessa forma a fim de separar os componentes de cada tela em classes diferentes, melhorando a legibilidade e o entendimento do código.

A classe `InterfaceMain` é a classe principal da camada de visão, agregando todas as demais classes que definem os *layouts* de interface. Assim como as demais classes `Composite`, ela também possui uma classe onde estão definidos os eventos dos componentes gráficos. Essa classe implementa as interfaces da camada de controle, sendo a partir dela efetuada a comunicação entre essas duas camadas.

As interfaces da camada de controle definem quais são as informações necessárias a serem configuradas e como essas informações devem ser passadas ao controlador. Dessa forma, a classe `InterfaceMain` implementa os métodos das interfaces, obtendo os dados informados pelo desenvolvedor, formatando essas informações quando necessário e passando-as ao controlador. Quando o controlador retornar alguma informação para ser exibida na tela, novamente é feita a chamada a um método de uma interface e a classe `InterfaceMain` formata essas informações para então exibi-las nas suas telas.

3.4.3 Operacionalidade da implementação

Nessa seção é apresentada a operacionalidade da ferramenta, descrevendo os passos para a construção das telas de cadastro e consulta de um sistema de locação de filmes. O projeto de banco de dados para o sistema de locação de filmes foi modelado de acordo com a figura 14.

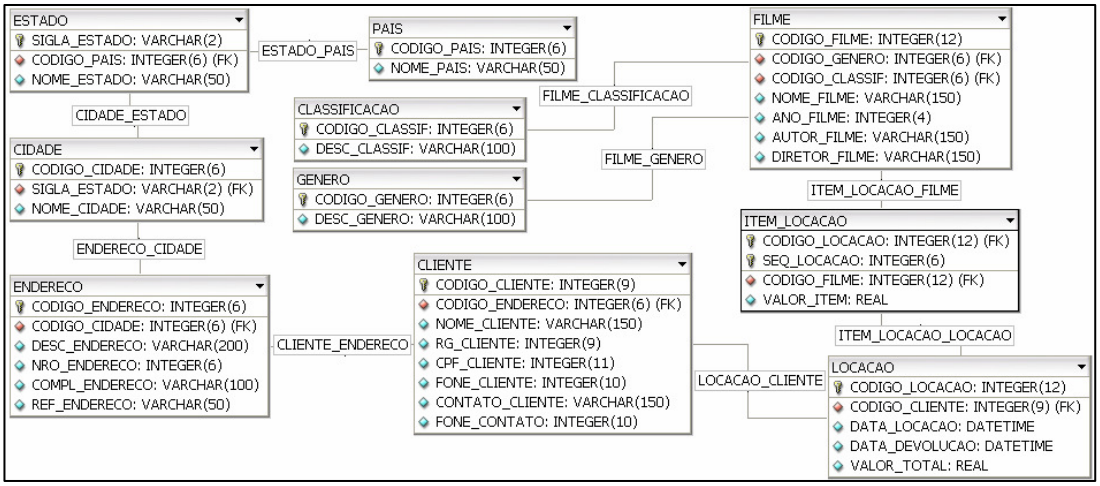


Figura 14 – Projeto de banco de dados para um sistema de locação de filmes

A figura 15 apresenta a tela principal da ferramenta, que é exibida ao desenvolvedor ao iniciar a execução da aplicação.

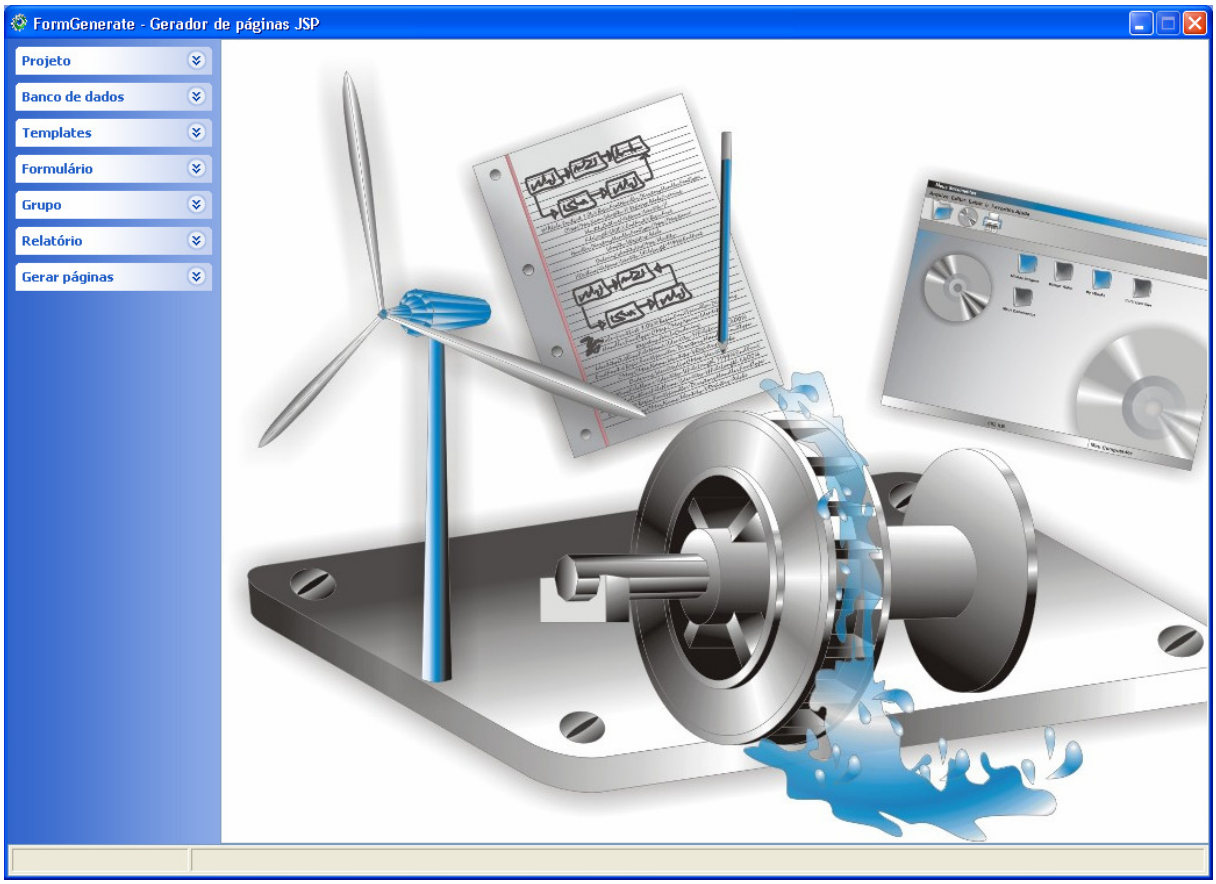


Figura 15 – Tela inicial do FormGenerate

Para efetuar a geração de código a partir de FormGenerate é necessário seguir alguns passos: configurar o projeto, obter as informações de entrada (extrair o metadados) e configurar formulários, grupos e relatórios. Algumas das informações a serem cadastradas são meros informativos ao desenvolvedor, já outras são de extrema importância para a geração de código, pois elas serão utilizadas no *merge* com os *templates*.

A figura 16 apresenta como efetuar a configuração do projeto.

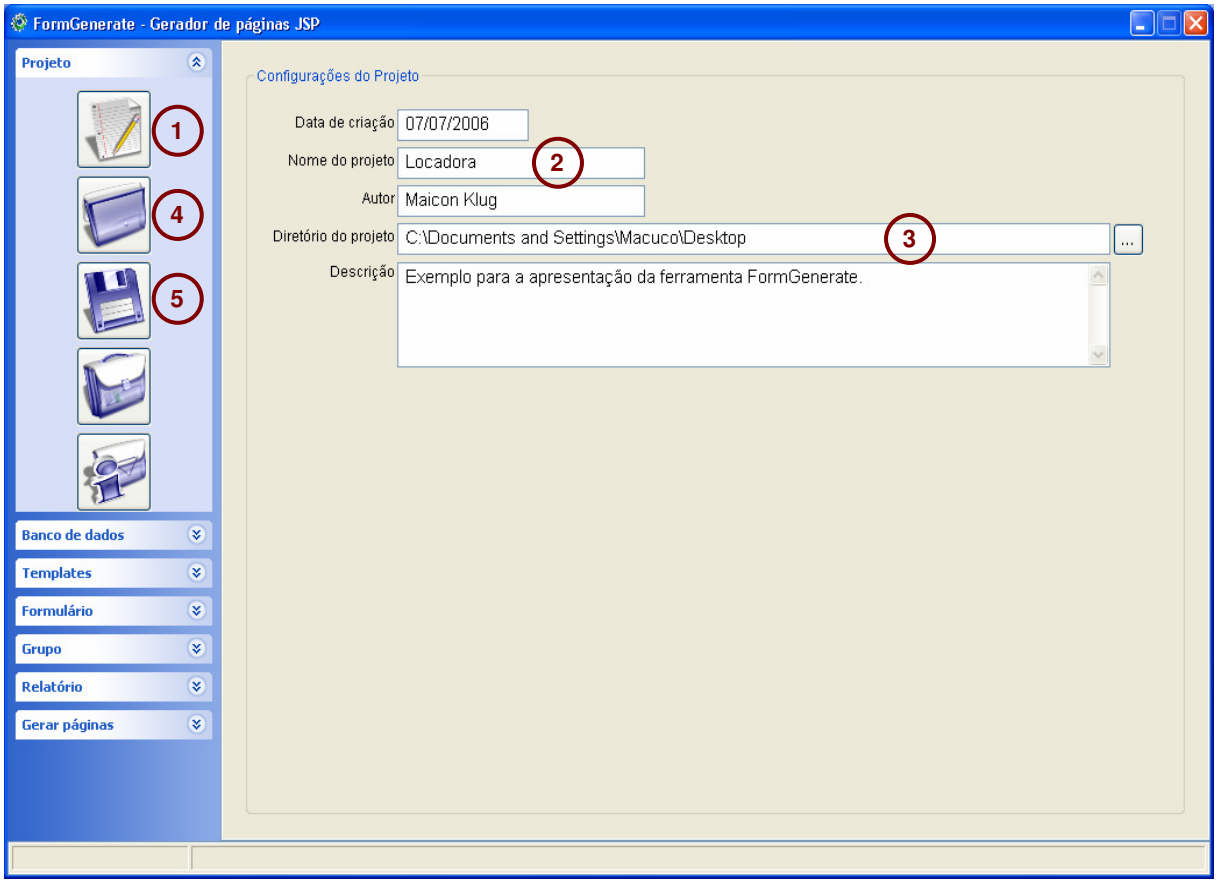


Figura 16 – Configurando um projeto

O primeiro passo a ser feito é carregar um projeto em memória, para isso é necessário criar um novo projeto (1) ou abrir um projeto já existente (4). Ao abrir um projeto, os campos da tela serão preenchidos automaticamente com as informações previamente cadastradas. Ao criar um novo projeto, é necessário preencher todas as informações, sendo que as principais são: o nome do projeto (2) e o local onde o projeto será salvo (3). Ao salvar as informações (5) é apresentada a mensagem conforme a figura 17.

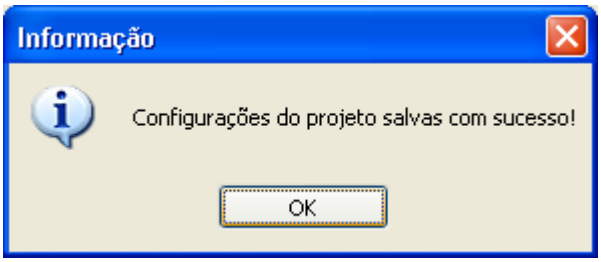


Figura 17 – Mensagem informando que as configurações do projeto foram salvas

Após o projeto estar configurado e salvo, é possível iniciar a configuração das informações de conexão com o banco de dados, conforme apresentado na figura 18.

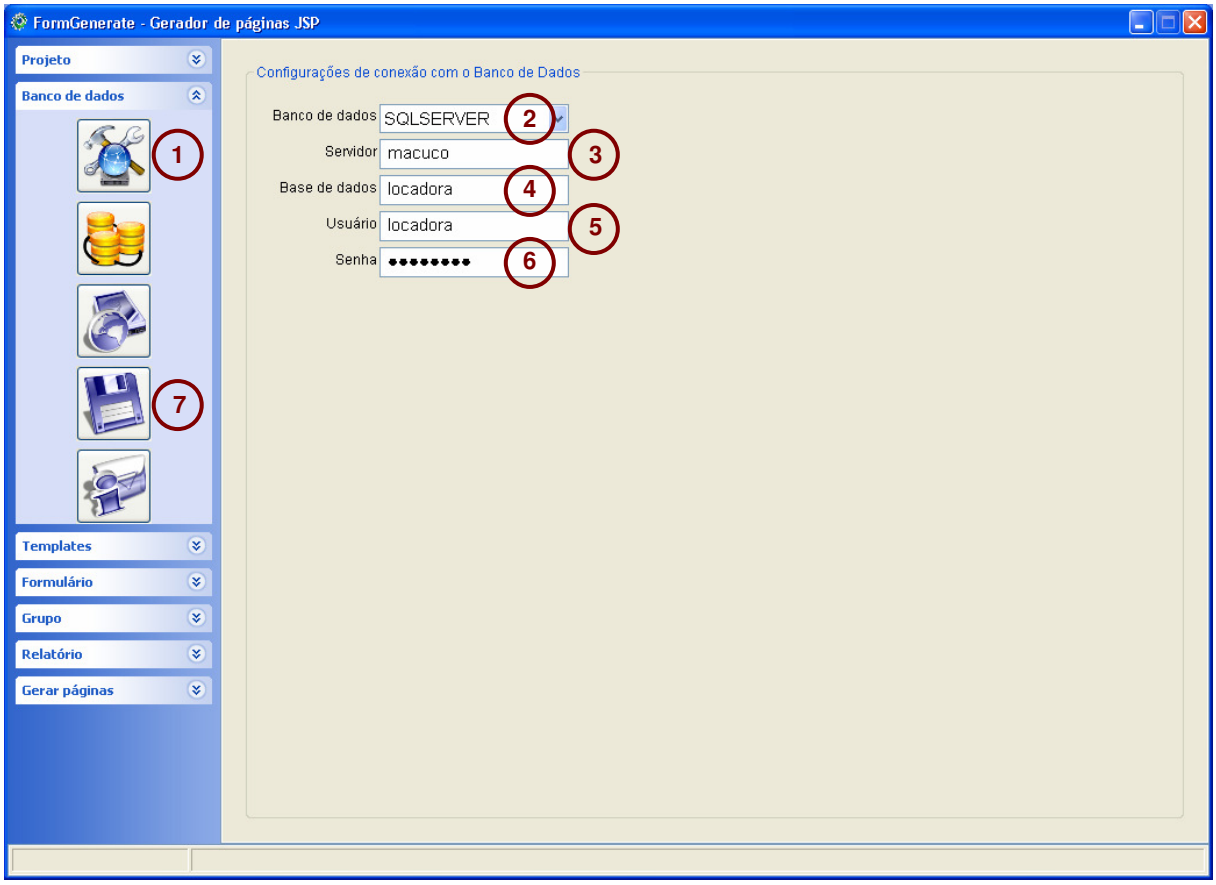


Figura 18 – Configurando a conexão com o banco de dados

A configuração das informações de conexão com o banco de dados é habilitada ao clicar no botão "Configurar conexão" (1). A partir desse momento, o desenvolver seleciona o banco de dados a ser utilizado (2), informa o nome do servidor de banco de dados onde a base de dados está hospedada (3), o nome da base de dados (4), o usuário para conexão (5) e a senha do usuário (6). Ao salvar as informações (7), é apresentada a mensagem conforme a figura 19.

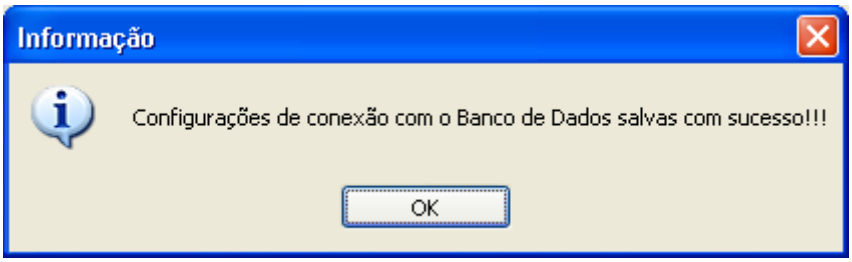


Figura 19 – Mensagem informando que as configurações de conexão com o banco foram salvas

Com as configurações de acesso ao banco de dados definidas, é possível extrair as informações do metadados conforme demonstrado na figura 20.

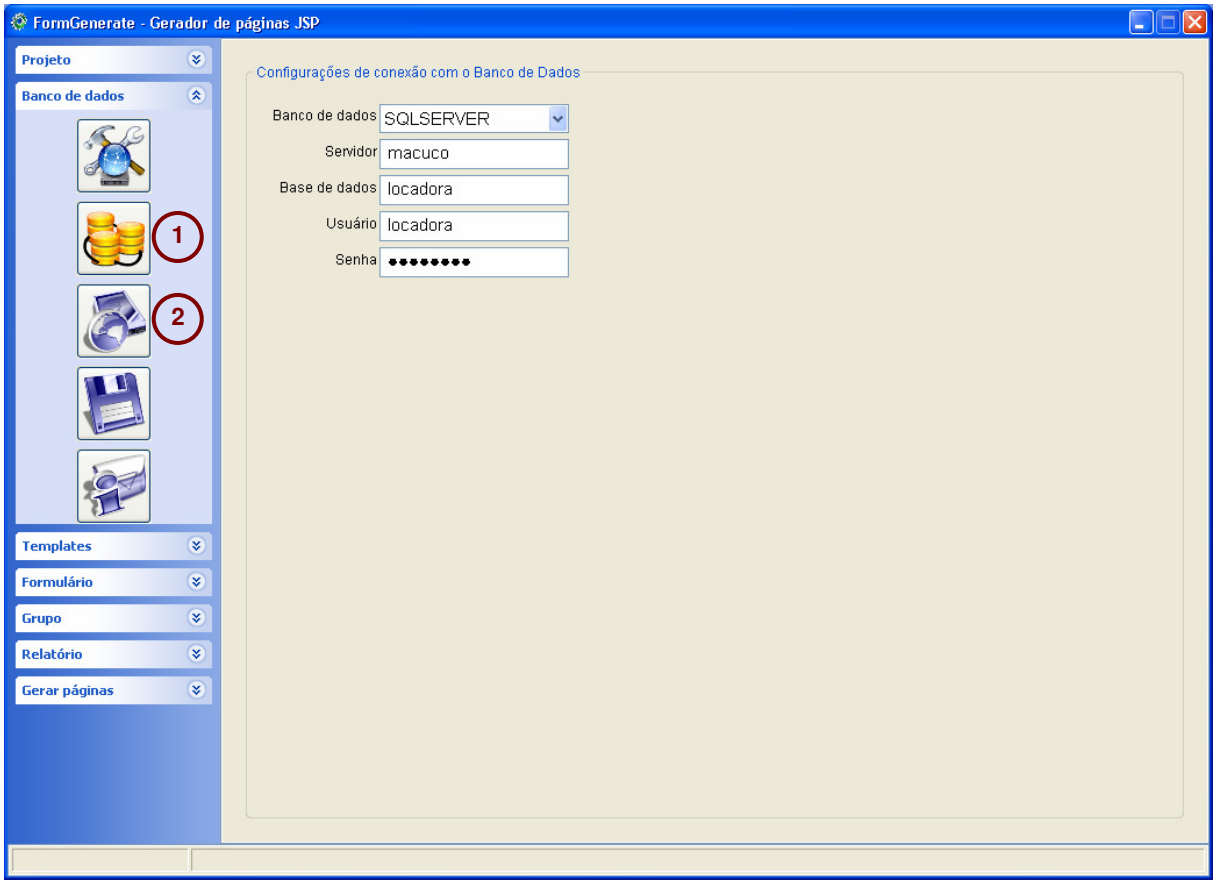


Figura 20 – Conectando e extraindo o metadados

A conexão com o banco de dados (1) pode ser feita antes ou após salvar as configurações, já a extração do metadados (2) é feita somente após estabelecida a conexão com o banco de dados.

Na configuração dos *templates* devem ser informados os arquivos modelos para a geração do código de saída. Cada *template* tem o seu propósito, não podendo ser informado o mesmo *template* para duas finalidades diferentes. A ferramenta faz essa verificação através do nome do *template*, ou seja, cada *template* deve possuir um nome diferente, mesmo que seus conteúdos sejam iguais. Todo o código de saída é gerado a partir dos *templates* utilizados. Quanto melhor codificado o *template*, melhor qualidade terá o código gerado. Para o desenvolvimento do *template*, é necessário que o desenvolvedor conheça os comandos e diretivas da VTL, para então conseguir referenciar, nos *templates*, os objetos gerados através das configurações efetuadas pela ferramenta. No apêndice 1 é apresentada a relação das classes utilizadas pelo Velocity para a geração de código. Objetos dessas classes são enviados pela ferramenta ao Velocity, o que permite ao desenvolvedor fazer chamadas aos métodos dessas classes na codificação dos *templates*.

A figura 21 apresenta os passos para a configuração dos *templates*.

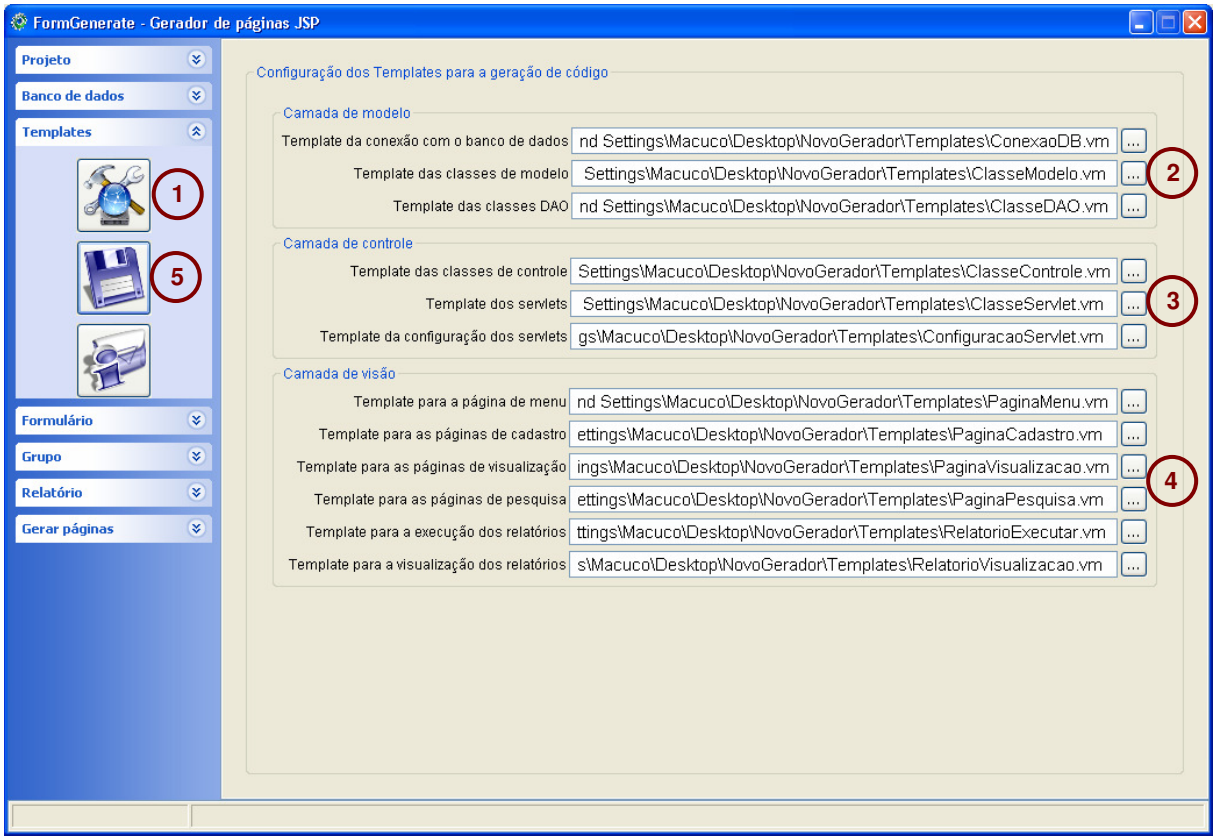


Figura 21 – Configurando os *templates*

A tela de configuração é habilitada ao clicar no botão "Configurar templates" (1). Logo após, devem ser selecionados os *templates* para a camada de modelo (2), para a camada de controle (3) e para a camada de visão (4). Ao salvar as configurações (5) é apresentada uma mensagem de processo efetuado com sucesso, conforme a figura 22.

Figura 22 – Mensagem informando que as configurações dos *templates* foram salvas

Depois de configurados o projeto, a conexão com o banco de dados e os *templates*, e extraído o metadados, inicia-se a configuração dos formulários. Um formulário é a representação de uma tela de cadastro para uma tabela do banco de dados. Esse processo é feito selecionando as tabelas e campos que serão apresentados nas telas e configurando os seus relacionamentos. A figura 23 demonstra como é realizada a configuração de um formulário no FormGenerate.

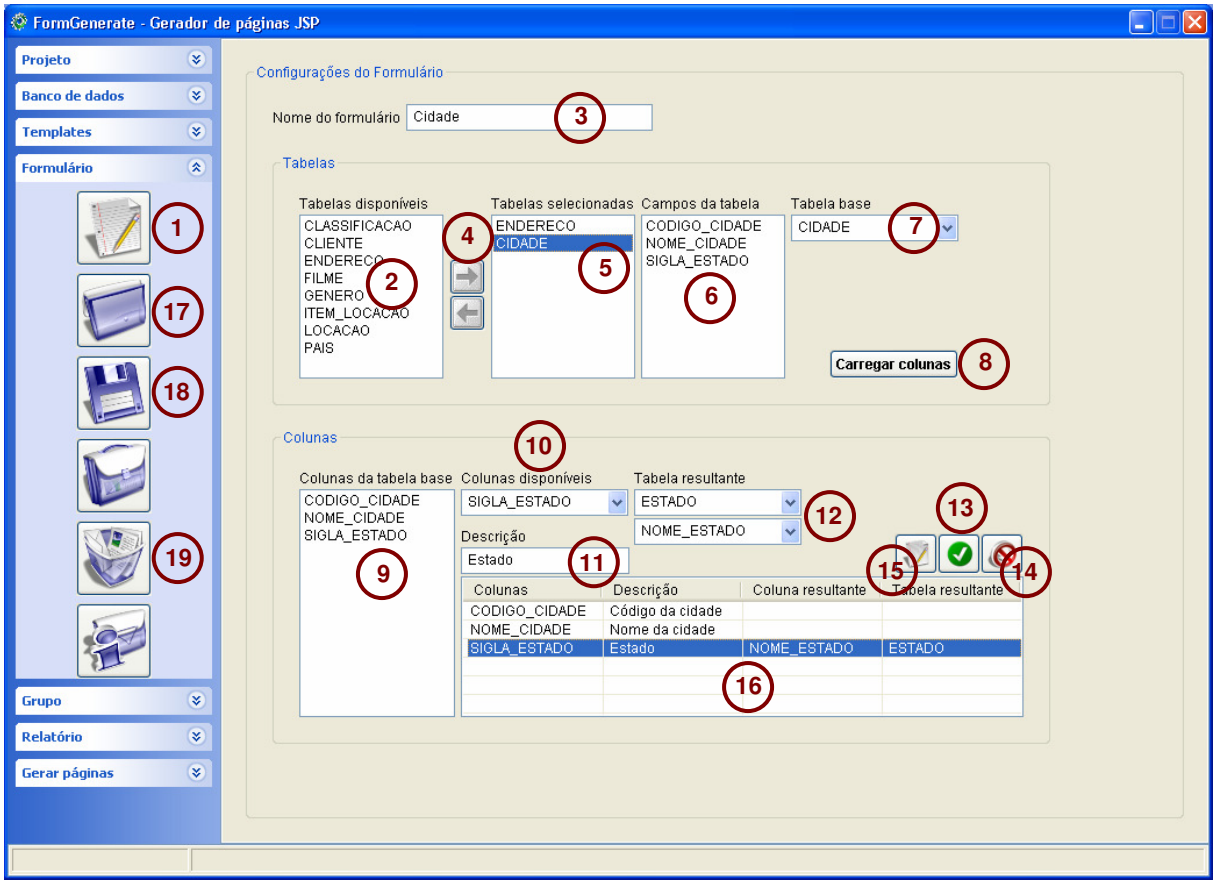


Figura 23 – Configurando formulários

Ao clicar no botão “Novo formulário” (1) é habilitada a tela de configurações de formulário e as tabelas disponíveis para seleção são carregadas (2) para dar início às parametrizações necessárias. O desenvolvedor então define um nome para a identificação do formulário (3) e seleciona a tabela (4) para qual será posteriormente gerada a respectiva tela de cadastro. Ao selecionar uma tabela que possua relacionamentos com outra(s) tabela(s), é apresentada uma mensagem informando ao usuário que as tabelas relacionadas também serão selecionadas, conforme mostrado na figura 24. Clicando sobre uma tabela selecionada (5), são apresentadas suas respectivas colunas (6). Dentre as tabelas selecionadas, o desenvolvedor seleciona a tabela que será a base do formulário (7), essa tabela será a tabela principal da página JSP no momento da geração de código. Com as tabelas selecionadas e a tabela base definida, o desenvolvedor passa para a configuração das colunas do formulário. Ao clicar no botão “Carregar colunas” (8), as colunas da tabela base são disponibilizadas para a configuração (9). Uma coluna do formulário é composta por: uma coluna da tabela base (10), uma descrição (11) e uma tabela e coluna resultante (12). A descrição é a identificação da coluna na página de cadastro a ser gerada e a tabela e coluna resultantes informam o relacionamento da tabela base com outra tabela.

Com as informações da coluna definidas, a mesma deve ser adicionada ao formulário

(13), sendo possível visualizá-la (16), juntamente com todas as colunas do formulário já configuradas. A ferramenta também permite a exclusão de uma coluna já inserida, bastando selecioná-la (16) e clicar no botão de exclusão (14). Pode-se também limpar os campos de configuração clicando no botão limpar (15).

A tela de configurações de formulários também possui recurso para abrir um formulário previamente cadastrado (17), carregando todas as informações do formulário para a tela. Ao salvar as configurações do formulário, clicando no botão "Salvar formulário" (18), é apresentada uma mensagem de sucesso (figura 25). Ao optar por excluir o formulário do projeto (19), uma mensagem informativa é apresentada ao usuário e caso o mesmo confirme, o arquivo com as definições do formulário é excluído. A mensagem de confirmação de exclusão do formulário é apresentada na figura 26.

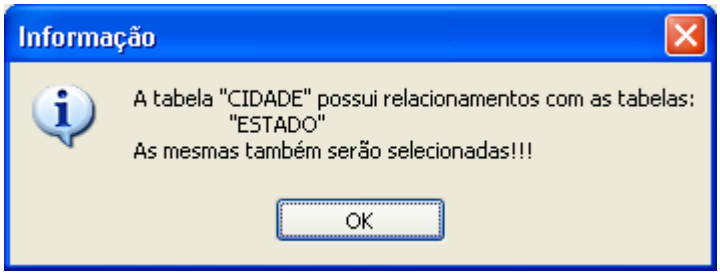


Figura 24 – Mensagem informando as tabelas selecionadas

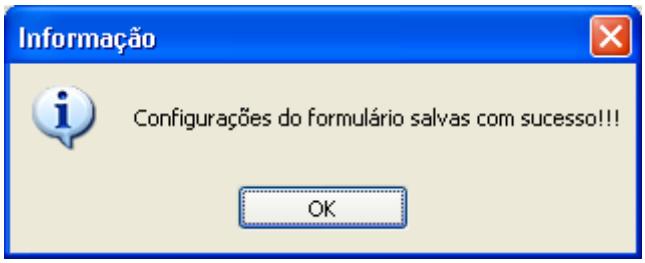


Figura 25 – Mensagem informando que as configurações do formulário foram salvas

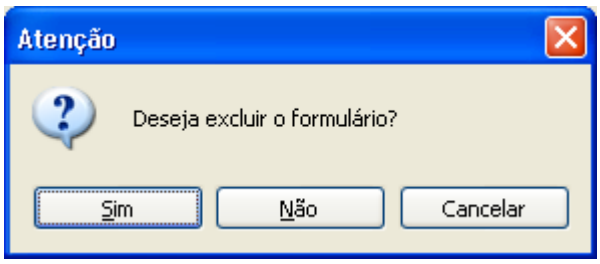


Figura 26 – Mensagem de confirmação de exclusão do formulário

Após efetuada a configuração dos formulários, é possível dar início à configuração dos grupos. Um grupo é um agrupamento de formulários e é utilizado na geração das páginas para representar o menu da aplicação a ser gerada automaticamente. Para a criação de um grupo é necessário que exista configurado no mínimo um formulário, caso contrário, a criação de grupos estará desabilitada. A figura 27 apresenta a tela de configuração de grupos.

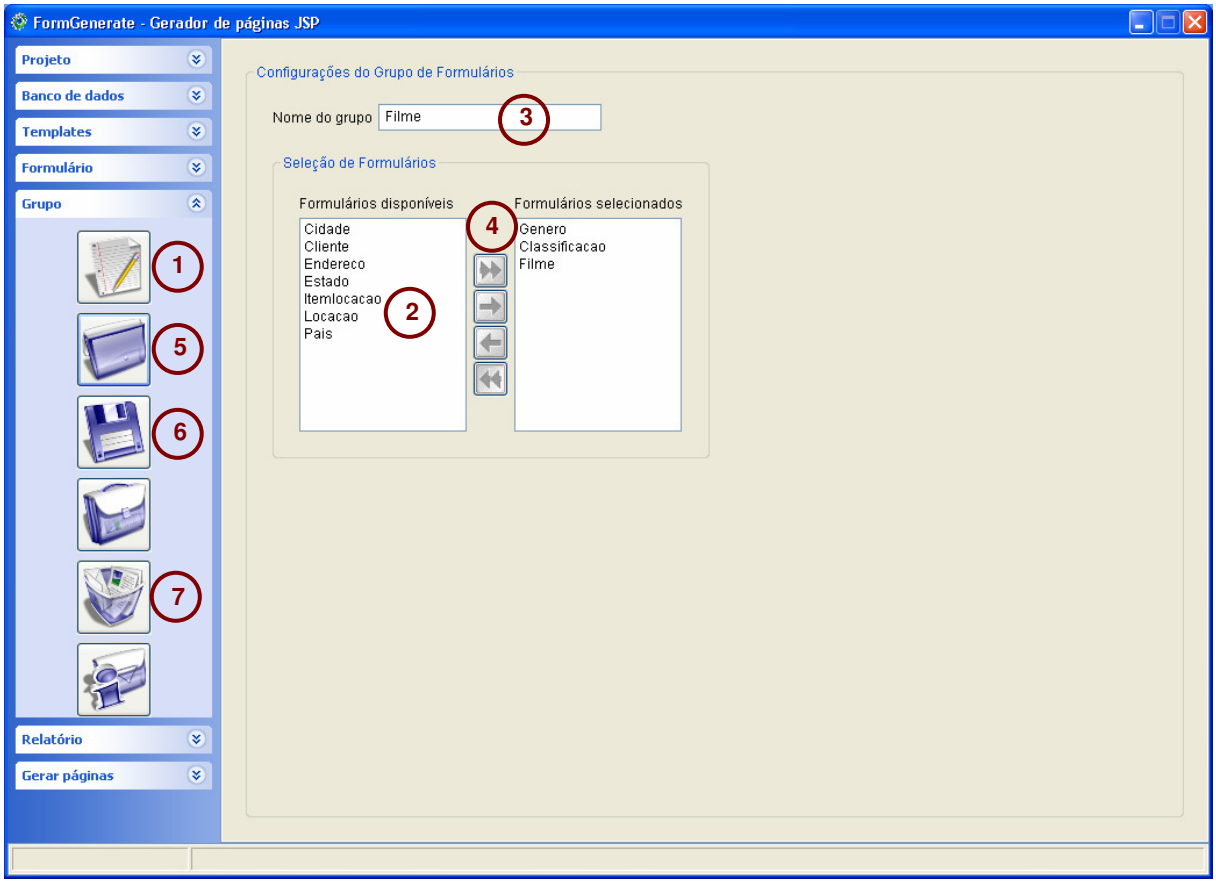


Figura 27 – Configurando grupos

Clicando no botão “Novo grupo” (1), a tela de configuração de grupos é habilitada e os formulários disponíveis para a seleção (2) são carregados para que o desenvolvedor inicie a configuração dos grupos. É necessário informar um nome de identificação para grupo (3) e selecionar os formulários que farão parte do grupo (4).

Ao abrir um grupo (5) previamente salvo, as configurações desse grupo são carregadas na tela, ficando como formulários disponíveis (2) todos os formulários do projeto exceto os que já estiverem selecionados. Clicando no botão “Salvar formulário” (6), as configurações do grupo são salvas e é apresentada uma mensagem de sucesso, conforme figura 28. Ao optar por excluir o formulário do projeto (7), uma mensagem informativa é apresentada ao usuário e caso o mesmo confirme, o arquivo com as definições do formulário é excluído.

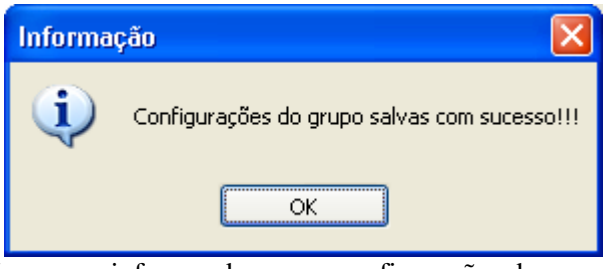


Figura 28 - Mensagem informando que as configurações do grupo foram salvas

A configuração dos relatórios é opcional, não influenciando diretamente no código a ser

gerado. Mas, caso sejam criados relatórios, serão geradas páginas específicas para a execução e visualização dos mesmos. A figura 29 demonstra a tela de configuração de relatórios.

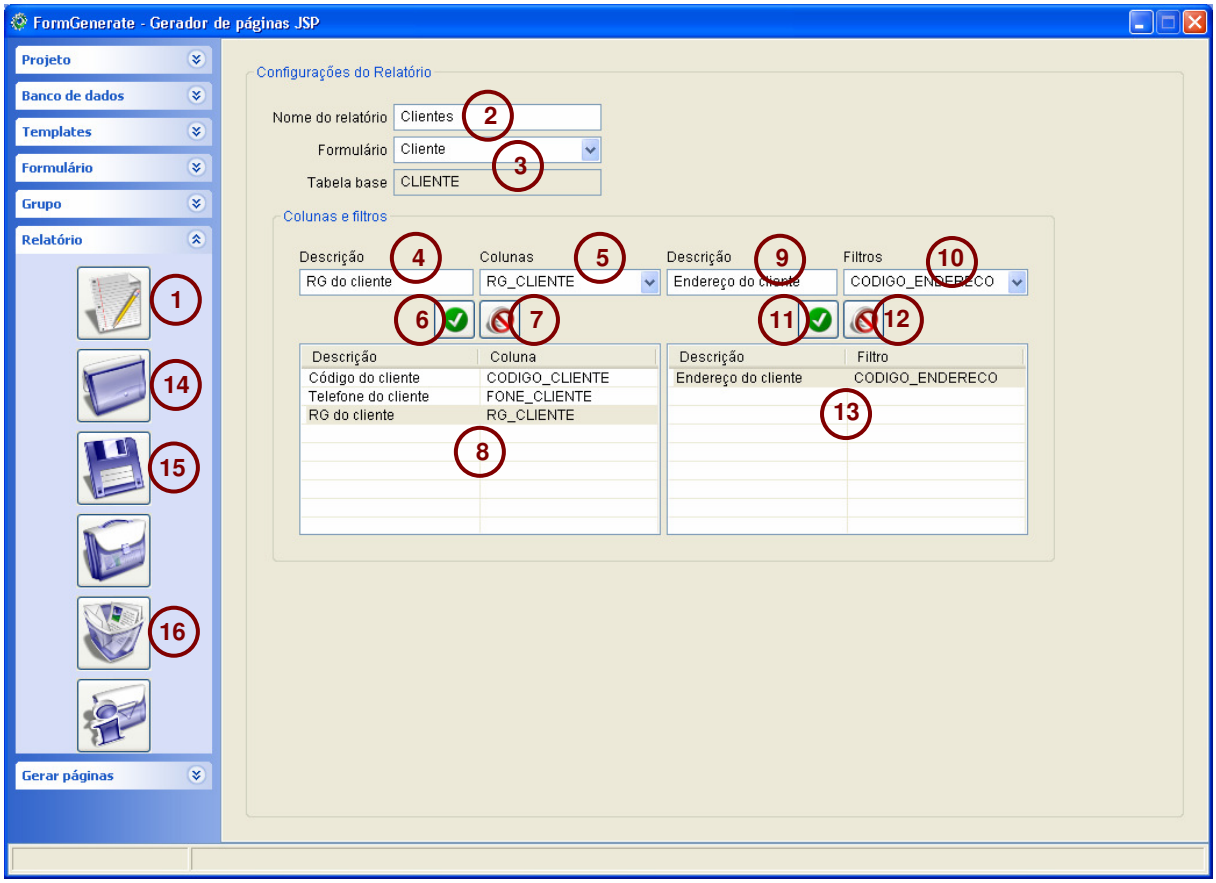


Figura 29 – Configurando relatórios

A configuração de um relatório é iniciada ao clicar no botão “Novo relatório” (1), sendo habilitada a tela de configurações e carregados os formulários do projeto. Para a criação de um relatório é necessário que exista ao menos um formulário cadastrado. Com a tela habilitada, o desenvolvedor informa um nome para o relatório (2) e seleciona qual o formulário será utilizado (3). Ao informar o formulário, é obtida a sua tabela base (3) e carregadas as suas colunas para a seleção de colunas (5) e seleção de filtros (10) do relatório.

Para a configuração das colunas do relatório, é necessário informar uma descrição (4) e uma coluna da tabela base do relatório (5). Para adicionar a coluna do relatório, basta clicar no botão adicionar (6). Assim a coluna configurada será apresenta na relação de colunas do relatório (8) juntamente com as demais colunas já configuradas. Para excluir uma coluna do relatório, é necessário selecioná-la (8) e clicar no botão remover (7).

Os filtros permitem ao usuário definir uma abrangência de informações, selecionando somente o que realmente ele necessita visualizar no relatório. A configuração dos filtros é feita da mesma forma que a configuração das colunas: informando uma descrição (9), selecionando uma coluna da tabela base (10), adicionando (11) ou removendo (12) filtros.

Todos os filtros criados são apresentados na relação de filtros do relatório (13). Mas essa configuração é opcional, permitindo ao desenvolvedor criar relatórios sem a definição de filtros.

É possível abrir um relatório previamente salvo, bastando clicar no botão “Abrir relatório” (14). Para salvar as definições de um relatório é necessário clicar no botão “Salvar relatório” (15). Nesse momento é apresentada uma mensagem informando ao usuário que as configurações efetuadas foram salvas (figura 30). Para excluir um relatório, o botão a ser clicado é o “Excluir relatório” (16).

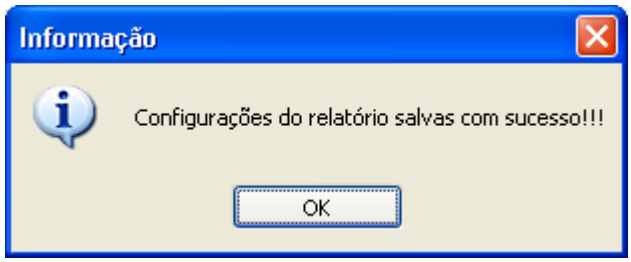


Figura 30 - Mensagem informando que as configurações do relatório foram salvas

Após efetuados todos os passos das configurações, pode ser dado início à geração de código, lembrando que todo o código gerado é definido em *templates*, ficando externo a ferramenta. A geração pode ser feita de duas maneiras: gerando cada camada individualmente ou todas as camadas de uma única vez. A geração individual pode ser utilizada enquanto os *templates* estão em fase de desenvolvimento, assim pode ser testado e avaliado o código gerado para a camada que se está se implementando. A geração de todas as camadas pode ser utilizada quando os *templates* já estão completamente prontos, onde o desenvolvedor está seguro do código a ser gerado. Os métodos que executam o processo de geração de código são os mesmos, independente de como o desenvolvedor optar por efetuá-lo, a diferença é que individualmente são chamados seus respectivos métodos e a geração completa faz a chamada dos métodos seqüencialmente. A figura 31 apresenta o menu que efetua a geração de código.

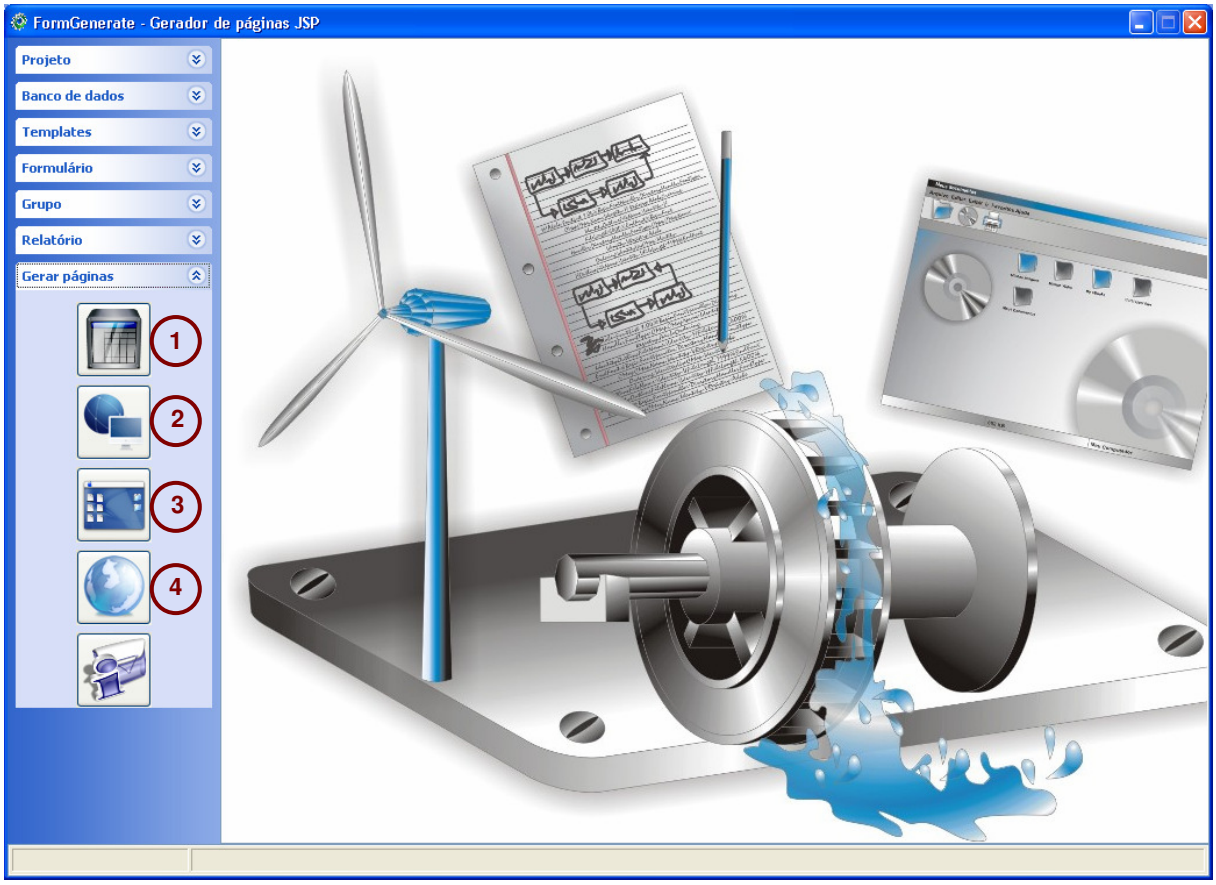


Figura 31 – Gerando código

A camada de modelo é gerada ao clicar no botão “Gerar camada de modelo” (1). Caso não ocorra nenhum erro durante a geração, é apresenta uma mensagem ao usuário informando que o código foi gerado com sucesso, conforme figura 32.

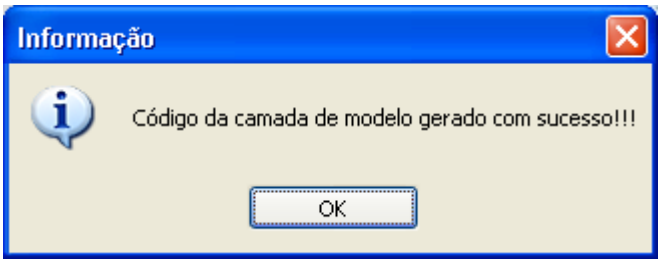


Figura 32 – Mensagem informando que a camada de modelo foi gerada

A camada de controle é gerada ao clicar no botão “Gerar camada de controle” (2). Caso não ocorra nenhum erro durante a geração, é apresenta uma mensagem ao usuário informando que o código foi gerado com sucesso, conforme figura 33.

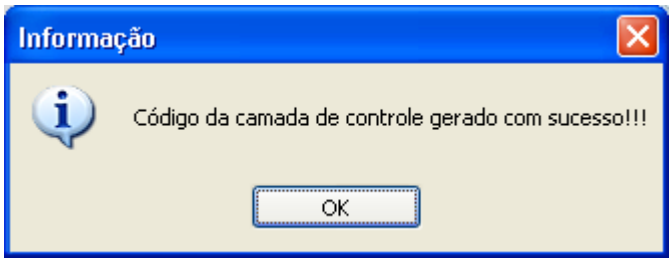


Figura 33 – Mensagem informando que a camada de controle foi gerada

A camada de visão é gerada ao clicar no botão “Gerar camada de visão” (3). Caso não ocorra nenhum erro durante a geração, é apresenta uma mensagem ao usuário informando que o código foi gerado com sucesso, conforme figura 34.

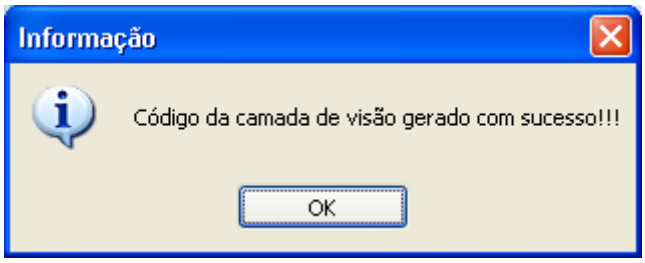


Figura 34 – Mensagem informando que a camada de visão foi gerada

Ao optar por “Gerar todas as camadas” (4), os processos de geração das camadas de modelo, controle e visão são executados seqüencialmente, e as mesmas mensagens são apresentadas ao desenvolvedor no término da geração de código de cada camada.

3.4.4 Código gerado

O FormGenerate não efetua a geração de código para uma aplicação completa, mas gera classes Java para as camadas de modelo e controle e páginas JSP para a camada de visão. As páginas JSP geradas possibilitam: o cadastro, alteração, exclusão e consulta das informações contidas na base de dados; e a execução de relatórios para a visualização dos dados utilizando filtros. As classes Java (através de classes de modelo, classes DAO, classes de controle e *servlets*) possibilitam a interação do código JSP das telas com o banco de dados. Para que aplicação se torne completa, é necessário desenvolver manualmente algumas páginas JSP para que seja efetuada a interação entre as páginas geradas. Mas, dependendo do nível de conhecimento do desenvolvedor na linguagem JSP, é possível que essa interação seja definida no próprio *template* utilizado para a geração das páginas.

Para que seja possível avaliar a consistência do código gerado e verificar se o mesmo funciona conforme o esperado, é preciso definir um ambiente para execução das telas e

relatórios gerados. O ambiente pode ser configurado manualmente, ou pode-se utilizar de ferramentas como o Eclipse ou o NetBeans. Independente da forma adotada, será necessário que exista um servidor de aplicações, como o Tomcat ou o Jboss, previamente instalado e configurado. No apêndice 2 é mostrado como criar um ambiente para a execução das páginas geradas utilizando o ambiente de desenvolvimento do Eclipse e o servidor de aplicações Tomcat.

As figuras 35 a 39 mostram as telas resultantes da geração de código para um sistema de locação de filmes, conforme modelagem apresentada na figura 14.

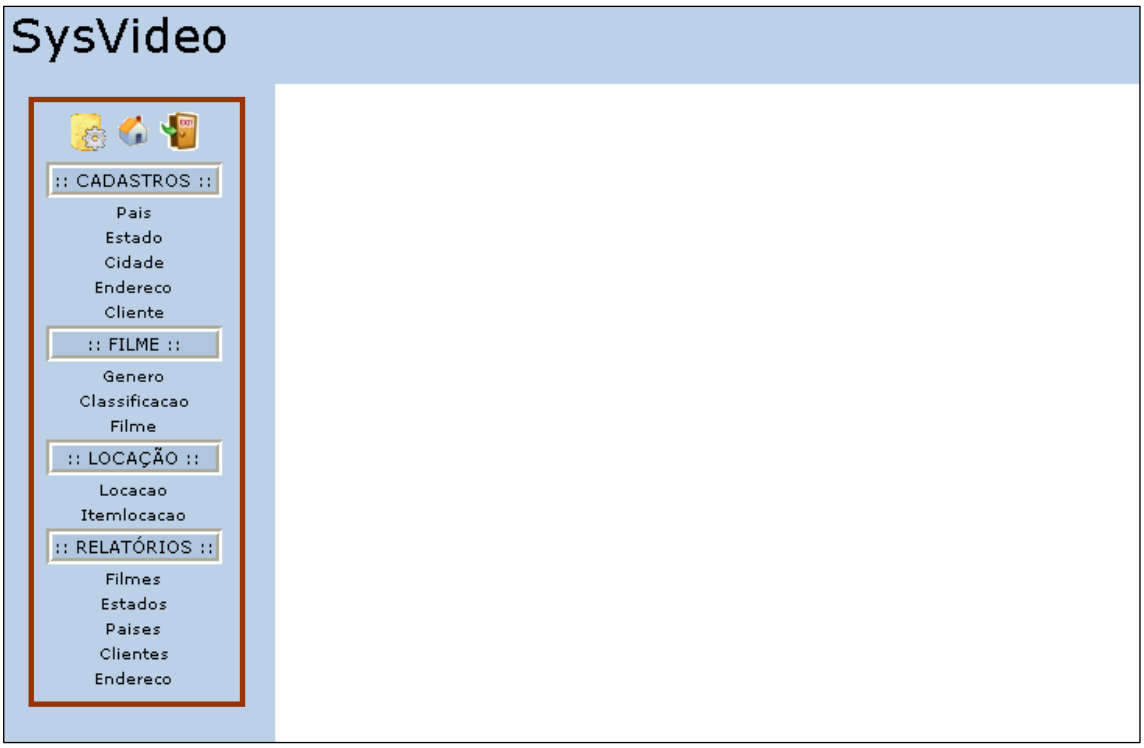


Figura 35 – Tela de menu

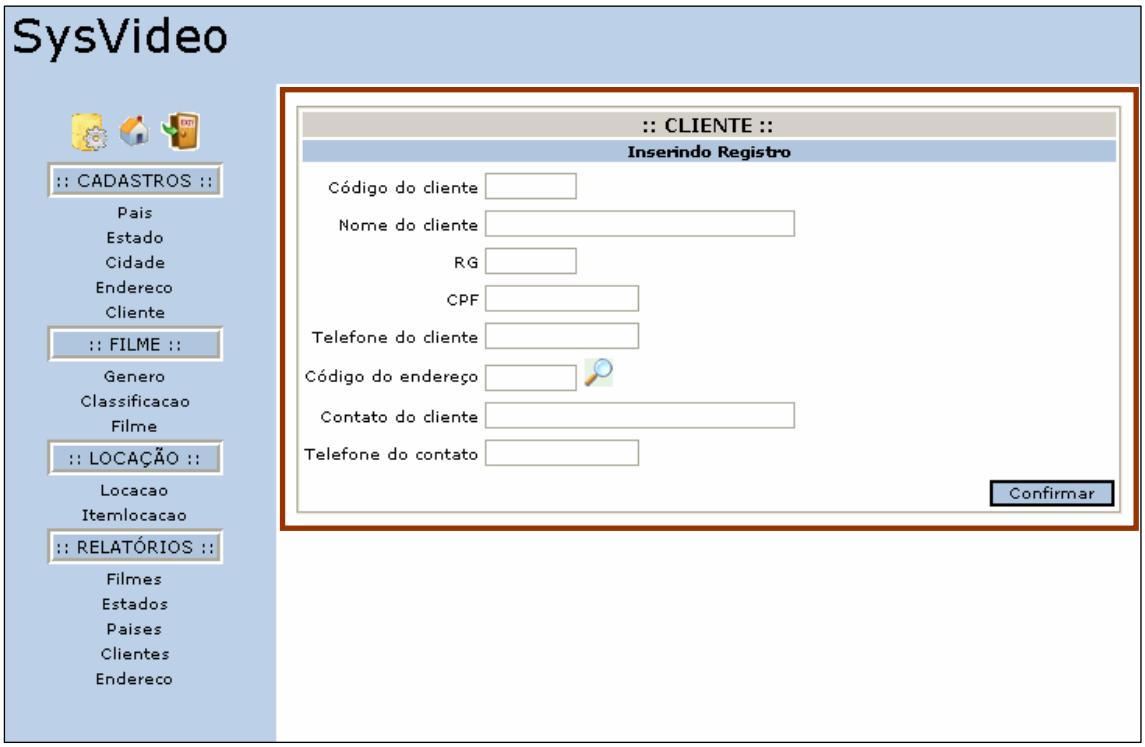
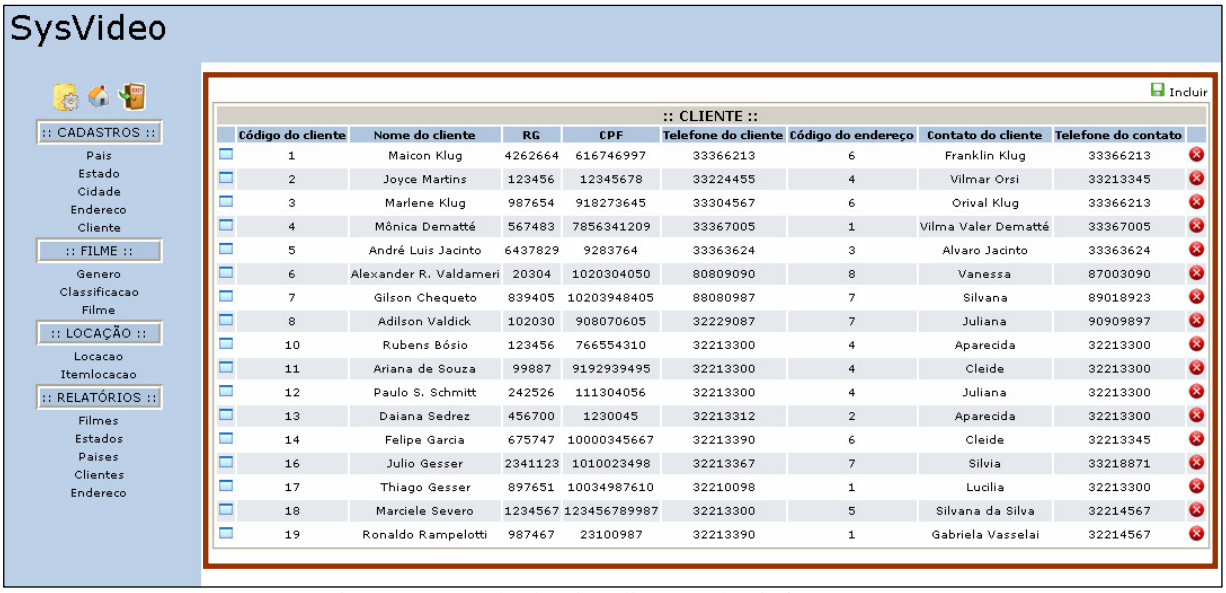


Figura 36 – Tela de cadastro e alteração das informações



Código do cliente	Nome do cliente	RG	CPF	Telefone do cliente	Código do endereço	Contato do cliente	Telefone do contato
1	Maicon Klug	4262664	616746997	33366213	6	Franklin Klug	33366213
2	Joyce Martins	123456	12345678	33224455	4	Vilmar Orsi	33213345
3	Marlene Klug	987654	918273645	33304567	6	Orival Klug	33366213
4	Mônica Dematté	567483	7856341209	33367005	1	Vilma Valer Dematté	33367005
5	André Luis Jacinto	6437829	9283764	33363624	3	Alvaro Jacinto	33363624
6	Alexander R. Valdameri	20304	1020304050	80809090	8	Vanessa	87003090
7	Gilson Chequeto	839405	10203948405	88080987	7	Silvana	89018923
8	Adilson Valdick	102030	908070605	32229087	7	Juliana	90909897
10	Rubens Bósio	123456	766554310	32213300	4	Aparecida	32213300
11	Ariana de Souza	99887	9192939495	32213300	4	Cleide	32213300
12	Paulo S. Schmitt	242526	111304056	32213300	4	Juliana	32213300
13	Daiana Sedrez	456700	1230045	32213312	2	Aparecida	32213300
14	Felipe Garcia	675747	10000345667	32213390	6	Cleide	32213345
16	Julio Gesser	2341123	1010023498	32213367	7	Silvia	33218871
17	Thiago Gesser	897651	10034987610	32210098	1	Lucilia	32213300
18	Marciele Severo	1234567	123456789987	32213300	5	Silvana da Silva	32214567
19	Ronaldo Rampelotti	987467	23100987	32213390	1	Gabriela Vasselai	32214567

Figura 37 – Tela de visualização das informações

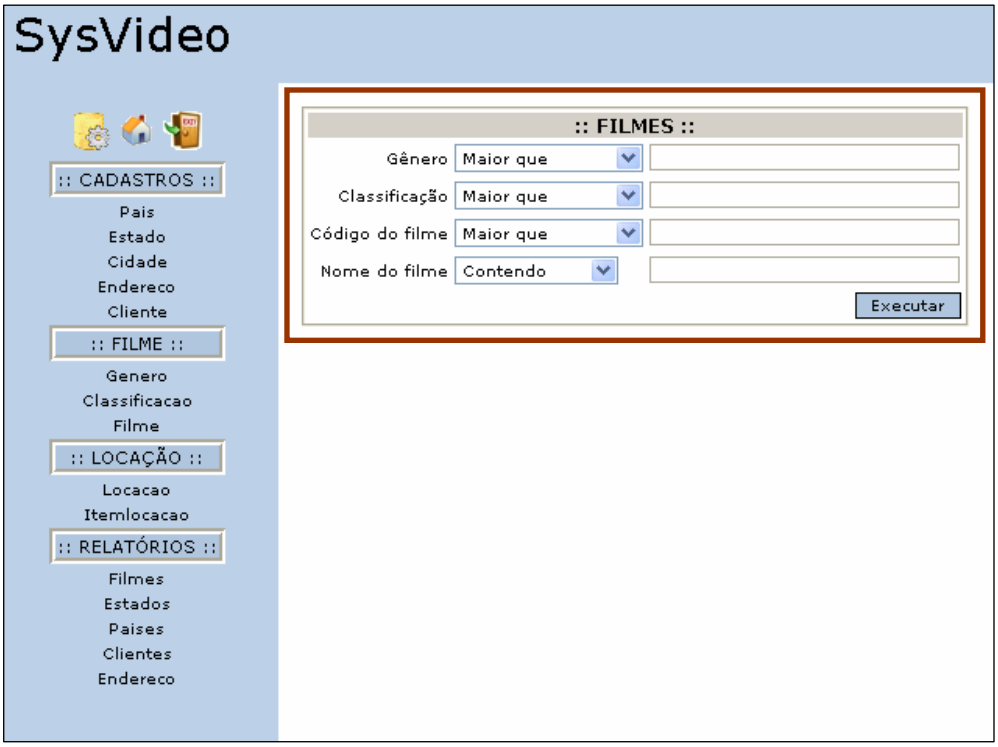


Figura 38 – Tela de execução de relatórios

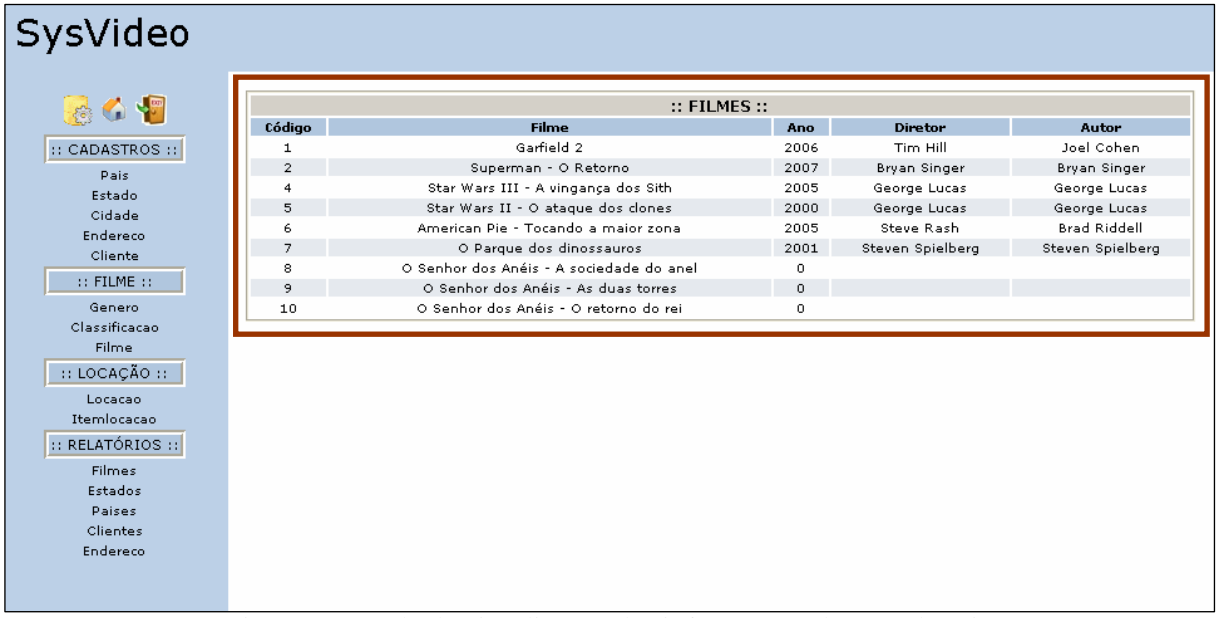


Figura 39 – Tela de visualização das informações de um relatório

3.5 RESULTADOS E DISCUSSÃO

O desenvolvimento da ferramenta apresentada alcançou o atendimento dos requisitos propostos. A implementação do FormGenerate foi baseada na ferramenta CodGer,

desenvolvida por Menin (2005), sendo mantidas algumas de suas funcionalidades, como: criação e configuração de projetos, configuração de conexão com o banco de dados, configuração de formulários e grupos e geração de páginas JSP. Em contrapartida, nenhum código fonte do CodGer foi reutilizado, pois o FormGenerate foi estruturado de maneira que: atendesse ao padrão MVC, efetuasse a conexão com outros SGBDs e pudesse utilizar *templates* para a geração de código.

A conexão com diferentes SGBDs foi possível através do uso da API JDBC e os respectivos *drivers* JDBC de cada SGBD elencado no item 1.1 deste trabalho. O CodGer também utiliza a API JDBC para acesso ao banco de dados, mas da forma que foi implementada a extração das informações do metadados, suporta somente ao MySQL.

No CodGer, o código das páginas JSP está fixo em seus fontes, o que permite a ferramenta gerar sempre a mesma saída. Essa restrição prejudica a legibilidade e a manutenção, tanto da ferramenta quanto do código gerado. No desenvolvimento do FormGenerate, foi utilizado o motor de *templates* Velocity, permitindo que todo o código gerado fique externo a aplicação. O uso de *templates* possibilita maior flexibilidade ao desenvolvedor, que pode criar os arquivos modelos que serão utilizados na geração de código, de acordo com a sua necessidade.

No quadro 26 é apresentada uma comparação entre o CodGer e o FormGenerate levando em consideração as suas principais características.

CARACTERÍSTICAS	CodGer	FormGenerate
desenvolvimento utilizando o conceito de orientação a objetos	sim	sim
desenvolvimento utilizando o padrão MVC	não	sim
uso de *templates* para geração de código	não	sim
utilização de *driver* JDBC para a conexão com o banco de dados	sim	sim
permite conexão com vários SGDBs	não	sim
utilização do motor de *templates* Velocity	não	sim
geração de código utilizando o padrão MVC	sim	sim

Quadro 26 – Comparação entre ferramentas CodGer e FormGenerate

Entre os trabalhos correlatos, verificou-se a existência de ferramentas com características similares ao da ferramenta apresentada nesse trabalho. O EasyCase também foi desenvolvido utilizando o padrão MVC, mas não utiliza o conceito de orientação a objetos

devido a sua integração com a ferramenta Oracle Designer6i. O TechCodeGenerate pode extrair o metadados de diferentes SGBDs e gera classes Java para as camadas de modelo e controle e código XSL para a camada de apresentação. Por fim, o ClassGenerator obtém as informações da estrutura de uma base de dados, e através de *templates* faz a geração de código nas linguagens PHP, Phyton e XHTML.

4 CONCLUSÕES

O FormGenerate foi desenvolvido com o objetivo de auxiliar o desenvolvedor na criação de rotinas de acesso ao banco de dados e de telas com funcionalidades de inclusão, alteração, exclusão e consulta de informações. A partir das informações extraídas de um metadados em um SGBD relacional, o desenvolvedor pode configurar telas, grupos de telas e relatórios para uma aplicação.

A API JDBC, utilizada para a extração das informações da estrutura da base de dados, é de fácil utilização e se mostrou uma excelente ferramenta para acesso ao banco de dados. Mesmo conectando em diferentes SGBDs, a API JDBC proporcionou uma padronização na implementação da rotina para extrair as informações do metadados, sendo que não foi preciso nenhum tratamento especial para particularidades dos bancos de dados. Para a conexão com o banco de dados Oracle XE e MYSQL 5.0.21 foram utilizados *drivers* JDBC específicos para cada um deles, já para acesso o Microsoft SQL Server 2000, foi utilizada a ponte JDBC-ODBC, disponibilizada com as demais APIs do JDK 1.5.0.

O uso de *templates* para a geração de código permitiu que o código gerado não ficasse totalmente dependente da ferramenta. A forma de codificar o *template* fica a critério do desenvolvedor, o qual deverá ter conhecimento da linguagem VTL do Velocity. Para a correta geração de código, os *templates* precisam ser definidos utilizando as diretivas e comandos da VTL. Para criar os *templates*, o desenvolvedor também precisa ter conhecimento dos objetos gerados pela ferramenta e seus métodos, para que possa obter os valores desses objetos dentro do *templates*, também através da VTL. No apêndice 1 são apresentados os tipos de objetos enviados para o contexto do Velocity.

A ferramenta foi desenvolvida adotando o padrão MVC, resultando em uma modularização distinta, coesa e pouco acoplada, permitindo que alterações em uma camada não afetem diretamente as outras camadas da aplicação.

O FormGenerate possibilita a conexão com três bancos de dados, e ao extrair as informações do metadados, trata somente os seguintes tipos de dado de colunas: caracter, cadeia de caracteres, data, números inteiros e números com casas decimais.

Na configuração dos formulários é definida uma tela de cadastro para uma tabela do banco de dados, não sendo possível inserir colunas de duas tabelas em uma mesma tela de cadastro, assim como também não é possível criar formulários mestre-detalhe.

Ainda na configuração de formulários, não é possível definir o tipo do componente

gráfico utilizado (por exemplo, `Edit`, `Label`, `Combo`, etc.) para representar a coluna na tela gerada. Essa configuração é determinada no *template*, assim como o tamanho do componente e a sua localização dentro da tela.

4.1 EXTENSÕES

Sugere-se como extensões para a ferramenta apresentada:

a) permitir que as informações de conexão com os SGBDs sejam obtidas a partir de um arquivo de propriedades ou um arquivo XML;
b) permitir a seleção dos diretórios onde serão salvos os arquivos gerados pelo ferramenta;
c) implementar na configuração das colunas do formulário a possibilidade de informar qual componente gráfico, na página gerada, corresponderá a coluna do formulário. Isso possibilitará maior flexibilidade no código do *template*, pois não será necessário verificar o tipo ou o tamanho do campo, simplesmente será verificado pelo nome ou pelo tipo de componente que deve representá-lo;
d) gerar uma versão do FormGenerate para ser executada em outro sistema operacional, como por exemplo Linux e MacOS. Observa-se que, como a ferramenta apresentada foi desenvolvida utilizando a API gráfica SWT, a mesma possibilita a sua utilização em diferentes sistemas operacionais;
e) implementar um módulo facilitador para a confecção dos *templates*.

REFERÊNCIAS BIBLIOGRÁFICAS

ABITEBOUL, S.; BUNEMAN, P.; SUCIU, D. **Gerenciamento de dados na web**. Tradução Mônica Cárdia. Rio de Janeiro: Campus, 2000.

APACHE SOFTWARE FOUNDATION. **Velocity**. [S.l.], 2004. Disponível em: <http://jakarta.aapache.org/velocity>. Acesso em: 19 maio 2007.

BEZERRA, E. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro: Campus, 2002.

DALGARNO, M. **Frequently asked questions about code generations**. [S.l.], 2006. Disponível em:<http://www.codegeneration.net/tiki-index.php?page=FrequentlyAskedQuestions>. Acesso em: 19 maio 2007.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 6. ed. Tradução Edson Furmankiewicz. São Paulo: Pearson, 2005.

FAGUNDES, E. M. **Quais são as diferenças entre os bancos de dados relacionais e os bancos de dados orientados a objetos?**[S.l.], 2007. Disponível em: <http://www.efagundes.com/artigos/Quais%20as%20diferencas%20entre%20os%20bancos%20de%20dados%20relacionais%20e%20os%20orientados%20a%20objetos.htm>. Acesso em: 19 maio 2007.

FRANCO, G. **A vida não é um padrão**. [S.l], 2005. Disponível em: <http://gamafranco.blogspot.com/2005/12/oracle-xe.html>. Acesso em: 09 jun. 2007.

GRAYSON, T. H. **Projeto de bancos de dados relacionais:** princípios de projetos de banco de dados. [S.l.], 2002. Disponível em: <http://www.universiabrasil.net/mit/11/11208/pdf/lecture5-2.pdf>. Acesso em: 19 maio 2007.

GUEDES, G. T. A. **UML 2**: guia de consulta rápida. 2. ed. São Paulo: Novatec, 2005.

HERRINGTON, J. **Code generation in action**. Greenwich: Manning, 2003.

JEVEAUX, P. C. M. **Aprenda a utilizar JDBC**. [S.l.], 2004. Disponível em: <http://portaljava.com/home/modules.php?name=Content&pa=list_pages_categories&cid=8>. Acesso em: 19 maio 2007.

JEVEAUX, P C. M.; MOURA, M D. **Velocity**. [S.l.], [2002?]. Disponível em: <http://www.portaljava.com.br/home//modules.php?name=News&file=article&sid=97>. Acesso em: 20 maio 2007.

LEAL, M. D. **ClassGenerator:** um gerador de artefatos multiplataforma. 2005. 76 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Federal do Pará, Belém. Disponível em: <http://www.marcelioleal.net/marcelio/sapp/conteudo/presentation/TccFinal.pdf>. Acesso em: 09 set. 2006.

MENIN, J. **Gerador de código JSP baseado em projeto de banco de dados MySQL**. 2005. 70 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MICROSOFT SQL Server. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Sqlserver>. Acesso em: 09 jun. 2007.

MOURA, M. F.; CRUZ, S. A. B. **Formatação de dados usando a ferramenta Velocity**. Campinas, 2002. Disponível em: <http://www.cnptia.embrapa.br/modules/tinycontent3/index.php?id=2>. Acesso em: 19 maio 2007.

MVC. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: <http://pt.wikipedia.org/wiki/MVC>. Acesso em: 14 jul. 2007.

MySQL. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: <http://pt.wikipedia.org/wiki/Mysql>. Acesso em: 09 jun. 2007.

RUMBAUGH, J. et al. **Modelagem e projetos baseados em objetos**. Tradução Dalton Conde de Alencar. Rio de Janeiro: Campus, 1994.

SANTOS, A. B. **Desenvolvimento de ferramenta e de processos para a geração automática de código Java a partir de um dicionário de dados**. 2005. 90 f. Trabalho de Conclusão (Bacharelado em Informática) – Centro de Ciências Exatas e Tecnológicas, Universidade do Vale do Rio dos Sinos, São Leopoldo. Disponível em: <http://www.inf.unisinos.br/alunos/arquivos/TC_AndersonBestteti.pdf>. Acesso em: 19 ago. 2006.

SCHVEPE, C. **Gerador de código Java a partir de arquivos do Oracle Forms 6i**. 2006. 76 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOBRAL, J. M. B. **Aula 4**: Java Database Connectivity. [Florianópolis], 2004. Disponível em: <http://www.inf.ufsc.br/~bosco/ensino/ine5625/slides/Aula04%20-%20JDBC.pdf>. Acesso em: 19 maio 2007.

STEPHENS, M. **Software reality**: automated code generation. [S.l.], 2002. Disponível em: <http://www.softwarereality.com/programming/code_generation4.jsp>. Acesso em: 19 maio 2007.

SUN MICROSYSTEMS. **Web-tier application framework design**. [S.l.], 2002a. Disponível em: <http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html>. Acesso em: 05 jul. 2007.

_____. **Java BluePrints:** Model-View-Controller. [S.l.], 2002b. Disponível em: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. Acesso em: 25 maio 2007.

_____. **Core J2EE patterns:** Data Access Object. [S.l.], 2002c. Disponível em: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>. Acesso em: 11 jul. 2007.

TAKECIAN, P. L. et al. **Sistema de gerenciamento de workflows**. 2004. 35 f. Trabalho de Formatura Supervisionado (Bacharelado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo. Disponível em: <http://gul.linux.ime.usp.br/~plt/mac499/monografia.pdf >. Acesso em: 19 maio 2007.

TECHNIQUE TI. **TechCodeGenerator**. [S.l.], 2003. Disponível em: <http://www.technique.com.br/servlet/com.technique.site.Controller?sid=Technique&command=engine&action=generator>. Acesso em: 22 ago. 2006.

TELEMACO, U. **O que é JSP**. [S.l.], 2005. Disponível em: <http://www.crieseuwebsite.com/artigos/artigo.php?categoria=jsp&id=1>. Acesso em: 21 maio 2007.

TELEMACO, U.; LIMA, G. **JavaBeans em JSP**. [S.l.], 2004. Disponível em <http://www.jeebrasil.com.br/mostrar/23>. Acesso em: 21 maio 2007.

APÊNDICE 1 – Classes enviadas ao Velocity para a geração de código

O FormGenerate envia vários objetos ao contexto do Velocity. Referenciando os métodos desses objetos através da VTL, é possível obter as informações necessárias para a geração do código de saída. No quadro 27 são mostradas as classes e os respectivos métodos que devem ser de conhecimento do desenvolvedor na criação dos *templates*.

Classe	Métodos

TableModel	`public String getName()`
	`public List getColumns()`
	`public PrimaryKeyModel getPk()`
	`public List getFks()`

ColumnModel	`public String getName()`
	`public int getPrecision()`
	`public int getSize()`
	`public String getType()`
	`public int getSeq()`
	`public String getOrder()`
	`public boolean isAllowNull()`

PrimaryKeyModel	`public String getName()`
	`public List getColumns()`
	`public TableModel getTable()`

ForeignKeyModel	`public String getName()`
	`public List getColumns()`
	`public TableModel getTable()`
	`public TableModel getTableTarget()`

ConnectionDBModel	`public String getDatabase()`
	`public String getDb()`
	`public String getPassword()`
	`public String getServer()`
	`public String getUser()`

FormModel	`public String getName()`
	`public List getColumnsForm()`
	`public List getColumnsTableBase()`
	`public TableModel getTableBase()`

ColumnFormModel	`public ColumnModel getColumnBase()`
	`public ColumnModel getColumnResult()`
	`public String getDescription()`
	`public TableModel getTableResult()`

GroupModel	`public String getName()` `public List getFormsSelecteds()`

ReportModel	`public List getColumnsReport()`
	`public List getColumnsFilter()`
	`public String getName()`
	`public TableModel getTableBase()`
	`public FormModel getForm()`

ColumnReportModel	`public ColumnModel getColumn()` `public String getDescription()`

Quadro 27 – Classes enviadas ao contexto do Velocity para a geração de código

APÊNDICE 2 – Montando ambiente para execução do código gerado

O ambiente de desenvolvimento do Eclipse possui uma integração com o servidor de aplicações Tomcat. Utilizando dessa integração, é possível executar páginas JSP dentro do Eclipse, simulando assim um *browser*. Utilizando esse recurso de integração do Eclipse e Tomcat, é possível executar e testar as páginas geradas pelo FormGenerate, afim de verificar se a geração de código foi efetuada corretamente.

Para criar e configurar o ambiente de teste, os seguintes passos devem ser seguidos:

a) criar um projeto web: deve-se acessar o menu *File/New/Other... /Web*, optar por *Dinamic Web Project*, informar um nome ao projeto e confirmar a operação. Será criado um novo projeto, conforme a figura 40;

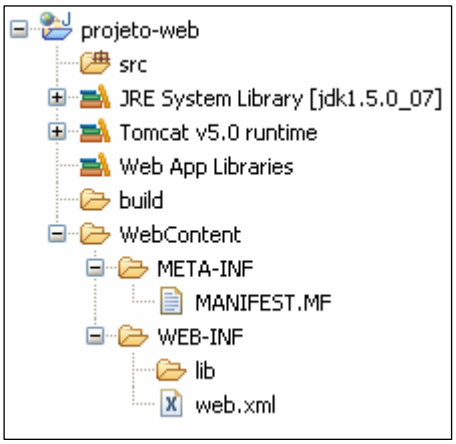


Figura 40 – Novo projeto web

b) configurar bibliotecas: é necessário copiar as bibliotecas `jstl.jar` e `standart.jar` para o diretório *WEB-INF/lib*. Caso a configuração do *Build Path* do projeto não seja feita automaticamente, será preciso clicar com o botão direito do mouse sobre o projeto web, ir na opção *Properties* e depois *Java Build Path*, selecionar a guia *Libraries*, clicar no botão *Add External JARs...* e selecionar os dois arquivos;

c) criar pacotes: devem ser criados os pacotes `controller`, `model`, `dao` e `servlet` dentro do *source forder* `src`. A estrutura de pacotes do projeto ficará conforme a figura 41;

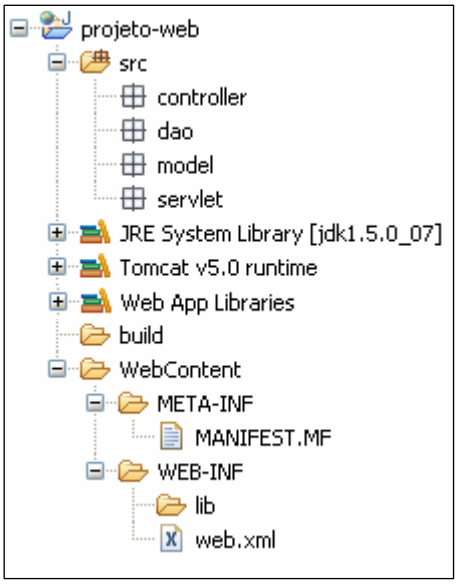


Figura 41 – Estrutura de pacotes do projeto web

d) copiar arquivos gerados para o projeto: os arquivos `.java` gerados pelo FormGenerate devem ser copiados para os respectivos pacotes no projeto web, os arquivos `.jsp` devem ser copiados para a pasta *WebContent* e o arquivo `web.xml` deve sobrescrever o arquivo da pasta *WEB-INF* (figura 42);

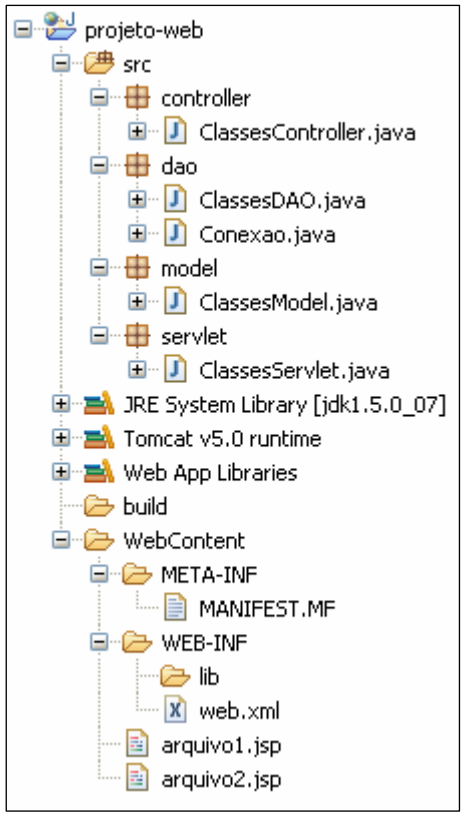


Figura 42 – Copiando arquivos gerados para o projeto web

e) criar páginas auxiliares: devem ser desenvolvidas páginas JSP para a página principal, navegação entre as páginas geradas, páginas de mensagem, páginas de erro, cabeçalho e outras. Foram desenvolvidas páginas para essas finalidades e o

código delas está disponível no apêndice 3.

Para efetuar todos os procedimentos acima, é necessário que o Eclipse possua os *plugins* para o desenvolvimento de aplicações na linguagem JSP. Também é necessário que o Tomcat esteja corretamente instalado e sua integração no Eclipse devidamente configurada.

APÊNDICE 3 – Páginas JSP auxiliares

Os quadros 28 a 30 apresentam páginas JSP que não são geradas automaticamente pela ferramenta, mas que são necessárias para a correta execução da aplicação. Essas páginas devem ser codificadas manualmente pelo desenvolvedor. O quadro 28 apresenta o código da página `index.jsp`, tela inicial da aplicação.

```
<html>
  <head>
    <frameset rows="65,*" frameborder="no" framespacing="0" border="0">
      <frame src="cabecalho.jsp" scrolling="no" noresize="noresize"
             name="cabecalho">
        <frameset cols="180,*" frameborder="no" framespacing="0" border="0">
          <frame src="Menu.jsp" scrolling="no" name="menu">
          <frame src="main.jsp" scrolling="yes" noresize="noresize" name="main">
        </frameset>
     </frameset>
   </head>
</html>
```

Quadro 28 – Página auxiliar `index.jsp`

O quadro 29 apresenta o código da página `cabecalho.jsp`, que exibe as informações no cabeçalho da aplicação gerada.

```
<html>
   <head>
      <link rel="stylesheet" type="text/css" href="folhaEstilo.css">
      <title>SysVideo - MySQL</title>
   </head>
   <body class="bodyMenu">
      SysVideo
   </body>
</html>
```

Quadro 29 – Página auxiliar `cabecalho.jsp`

O quadro 30 apresenta o código da página `erro.jsp`, que exibe os erros que possam vir a acontecer durante a execução da aplicação.

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="folhaEstilo.css">
    <script src="scripts.js"></script>
  </head>
  <body>
    <table class="tableForm" align="center" width="100%">
      <%String erro = request.getParameter( "erro" );%>
      <tr><td align="center" class="tituloTabela"><b><%=erro%></b></td></tr>
    </table>
  </body>
</html>
```

Quadro 30 – Página auxiliar `erro.jsp`

APÊNDICE 4 – *Template* para as classes de modelo

```
package model;

public class $convert.convertUpperFirst($tableModel.getName())Model {

   #foreach($column in $tableModel.getColumns())
   private $convert.convertType($column.getType(), $column.getPrecision())
   $convert.convertLowerAll($column.getName());
   #end

   public $convert.convertUpperFirst($tableModel.getName())Model() {
      super();
      #foreach($column in $tableModel.getColumns())
      #set($type = $convert.convertType($column.getType(),
           $column.getPrecision()))
      #if($type == "String")
      this.$convert.convertLowerAll($column.getName()) = "";#end
      #if($type == "char")
      this.$convert.convertLowerAll($column.getName()) = ' ';#end
      #if($type == "long")
      this.$convert.convertLowerAll($column.getName()) = 0;#end
      #if($type == "double")
      this.$convert.convertLowerAll($column.getName()) = 0.0;#end
      #if($type == "java.util.Date")
      this.$convert.convertLowerAll($column.getName()) = null;#end
      #end
   }
   #set($counter = 0)
   public $convert.convertUpperFirst($tableModel.getName())Model(
   #foreach($column in $tableModel.getColumns()) #set($counter = $counter + 1)
   #if ($counter == $tableModel.getColumns().size())
   $convert.convertType($column.getType(), $column.getPrecision())
   $convert.convertLowerAll($column.getName()))
   #else
   $convert.convertType($column.getType(), $column.getPrecision())
   $convert.convertLowerAll($column.getName()),
   #end #end {
      super();
      #foreach($column in $tableModel.getColumns())
      this.$convert.convertLowerAll($column.getName()) =
      $convert.convertLowerAll($column.getName());
      #end
   }

   #foreach($column in $tableModel.getColumns())
   public void set$convert.convertUpperFirst($column.getName())(
      $convert.convertType($column.getType(), $column.getPrecision())
      $convert.convertLowerAll($column.getName())) {
      this.$convert.convertLowerAll($column.getName()) =
      $convert.convertLowerAll($column.getName());
   }

   public $convert.convertType($column.getType(), $column.getPrecision())
      get$convert.convertUpperFirst($column.getName())() {
      return this.$convert.convertLowerAll($column.getName());
   }
   #end
}
```

Quadro 31 – *Template* para as classes de modelo

APÊNDICE 5 – *Template* para as classes DAO

```
#set($nomeClasse = ${convert.convertUpperFirst($tableModel.getName())})
#set($atributoClasse = ${tableModel.getName().toLowerCase()})
#set($tabela = ${tableModel.getName()})
#macro(montaComando $column $count)
#set($getColumnName = ${convert.convertUpperFirst($column.getName())})
#set($type = $convert.convertType($column.getType(), $column.getPrecision()))
#if($type == "String")pstmt.setString($count,
${atributoClasse}Model.get$getColumnName());#end
#if($type == "char")pstmt.setString($count,
Character.toString(${atributoClasse}Model.get$getColumnName()));#end
#if($type == "long")pstmt.setLong($count,
${atributoClasse}Model.get$getColumnName());#end
#if($type == "double")pstmt.setDouble($count,
${atributoClasse}Model.get$getColumnName());#end
#if($type == "java.sql.Date")pstmt.setDate($count,
${atributoClasse}Model.get$getColumnName());#end#end
#macro(montaInsert)#set($count = 0)
#foreach($column in $tableModel.getColumns())#set($count = $count + 1)
#if($count == ${tableModel.getColumns().size()})${column.getName()}"
#else${column.getName()},#end#end#end
#macro(montaUpdate)#set($count = 0)
#foreach($column in $tableModel.getColumns())#set($count = $count + 1)
#if($count == ${tableModel.getColumns().size()}) ${column.getName()} = ?"
#else ${column.getName()} = ?,#end#end#end
#macro(montaValores)#set($count = 0)
#foreach($column in $tableModel.getColumns())#set($count = $count + 1)
#if($count == ${tableModel.getColumns().size()})?)" );
#else?,#end#end#end
#macro(montaWhere)#set($count = 0)
#foreach($column in $tableModel.getPk().getColumns())#set($count = $count + 1)
#set($columnName = $column.getName().toUpperCase())
#if($count == ${tableModel.getPk().getColumns().size()}) $columnName = ?" );
#else $columnName = ?,#end#end#end
#macro(montaWhereSelect)#set($count = 0)
#foreach($column in $tableModel.getPk().getColumns())#set($count = $count + 1)
#set($columnName = $column.getName().toUpperCase())
#if($count == ${tableModel.getPk().getColumns().size()}) $columnName = ?" );
#else $columnName = ? AND#end#end#end
#macro(montaResultSet $column)
#set($type = "$convert.convertType($column.getType(), $column.getPrecision())")
#set($getColumnName = ${convert.convertUpperFirst($column.getName())})
#if($type == "String")${atributoClasse}Model.set${getColumnName}(
select.getString( "$column.getName()" ) );#end
#if($type == "char")${atributoClasse}Model.set${getColumnName}( select.getString(
"$column.getName()" ) );#end
#if($type == "long")${atributoClasse}Model.set${getColumnName}( select.getLong(
"$column.getName()" ) );#end
#if($type == "double")${atributoClasse}Model.set${getColumnName}(
select.getDouble( "$column.getName()" ) );#end
#if($type == "java.sql.Date")${atributoClasse}Model.set${getColumnName}(
select.getDate( "$column.getName()" ) );#end#end
#macro(montaParametros)#set($count = 0)
#foreach($column in $tableModel.getPk().getColumns())#set($count = $count + 1)
#set($columnName = $column.getName().toLowerCase())
#set($type = $convert.convertType($column.getType(), $column.getPrecision()))
#if($count == $tableModel.getPk().getColumns().size()) $type $columnName
#else $type $columnName,#end#end#end
#macro(montaComandoSelect $column $count)
#set($columnName = $column.getName().toLowerCase())
#set($type = $convert.convertType($column.getType(), $column.getPrecision()))
#if($type == "String")pstmt.setString($count, $columnName);#end
```

```
#if($type == "char")pstmt.setString($count, Character.toString($columnName));#end
#if($type == "long")pstmt.setLong($count, $columnName);#end
#if($type == "double")pstmt.setDouble($count, $columnName);#end
#if($type == "java.sql.Date")pstmt.setDate($count, $columnName);#end#end
public class ${nomeClasse}DAO {
   private Conexao conexao;
   public ${nomeClasse}DAO() throws Exception {
      this.conexao = Conexao.getInstanceOf();
   }
   public void insert(${nomeClasse}Model ${atributoClasse}Model) throws
            SQLException {
      if ( ${atributoClasse}Model != null ) {
         PreparedStatement pstmt = this.conexao.setPreparedStatement( "INSERT
          INTO ${tabela} (#montaInsert() + ") values ( #montaValores()
         #set($count = 0)
         #foreach($column in $tableModel.getColumns())#set($count=$count+1)
         #montaComando($column $count)#end
         this.conexao.execSQL(pstmt);
      }
   }
   public void update(${nomeClasse}Model ${atributoClasse}Model) throws
           SQLException {
      if ( ${atributoClasse}Model != null ) {
         PreparedStatement pstmt = this.conexao.setPreparedStatement( "UPDATE
         ${tabela} SET#montaUpdate()+" WHERE#montaWhere()
         #set($count = 0)
         #foreach($column in $tableModel.getColumns())#set($count=$count+1)
         #montaComando($column $count)#end
         #foreach($column in $tableModel.getPk().getColumns())
         #set($count = $count + 1)
         #montaComando($column $count)#end
         this.conexao.execSQL(pstmt);
       }
   }
   public void delete(${nomeClasse}Model ${atributoClasse}Model) throws
               SQLException {
      if ( ${atributoClasse}Model != null ) {
         PreparedStatement pstmt = this.conexao.setPreparedStatement( "DELETE
                                   $tabela WHERE#montaWhere()
         #set($count = 0)
         #foreach($column in $tableModel.getPk().getColumns())
         #set($count=$count+1)
         #montaComando($column $count)#end
         this.conexao.execSQL(pstmt);
      }
   }
   public ${nomeClasse}Model get${nomeClasse}Model( #montaParametros() ) throws
        SQLException {
      ${nomeClasse}Model ${atributoClasse}Model = null;
      PreparedStatement pstmt = this.conexao.setPreparedStatement( "SELECT * FROM
                                $tabela WHERE#montaWhereSelect()
      #set($count = 0)
      #foreach($column in $tableModel.getPk().getColumns())
      #set($count = $count + 1)
      #montaComandoSelect($column $count)
      #end
      ResultSet select = this.conexao.openSQL(pstmt);
      while ( select.next() ) {
         ${atributoClasse}Model = new ${nomeClasse}Model();
         #foreach($column in $tableModel.getColumns())
         #montaResultSet($column)#end
      }
      select.close();
      pstmt.close();
      return ${atributoClasse}Model;
   }
```

```
   public List select() throws SQLException {
      List<${nomeClasse}Model> objects = new ArrayList<${nomeClasse}Model>();
      ${nomeClasse}Model ${atributoClasse}Model = null;
      PreparedStatement pstmt = this.conexao.setPreparedStatement( "SELECT * FROM
                                $tabela ORDER BY 1" );
      ResultSet select = this.conexao.openSQL( pstmt );
      while ( select.next() ) {
         ${atributoClasse}Model = this.carregaDados( select );
         objects.add( ${atributoClasse}Model );
      }
      select.close();
      return objects;
   }
   public List select(List types, List params, List values, List operators )
               throws SQLException {
      List<${nomeClasse}Model> objects = new ArrayList<${nomeClasse}Model>();
      ${nomeClasse}Model ${atributoClasse}Model = null;
      String sql = "SELECT * FROM $tabela WHERE ";
      for ( int i = 0; i < params.size(); i++ ) {
         if ( i == params.size() - 1 )
         sql+=params.get(i)+getOperatorSelect((String)operators.get(i));
         else
         sql+=params.get(i)+getOperatorSelect((String)operators.get(i))+" AND ";
      }
      PreparedStatement pstmt = this.conexao.setPreparedStatement( sql );
      for ( int i = 0; i < values.size(); i++ ) {
         if(types.get(i).equals("String")||types.get(i).equals("char") )
            pstmt.setString( i + 1, (String)values.get(i));
         if(types.get(i).equals("long"))
            pstmt.setLong(i + 1, Long.parseLong((String)values.get(i)));
         if(types.get(i).equals("double"))
            pstmt.setDouble(i + 1, Double.parseDouble((String)values.get(i)));
         if (types.get(i).equals("java.sql.Date"))
            pstmt.setDate(i + 1, java.sql.Date.valueOf((String)values.get(i)));
      }
      ResultSet select = this.conexao.openSQL( pstmt );
      while ( select.next() ) {
         ${atributoClasse}Model = this.carregaDados( select );
         objects.add( ${atributoClasse}Model );
      }
      select.close();
      pstmt.close();
      return objects;
   }
   private ${nomeClasse}Model carregaDados( ResultSet select ) throws
              SQLException {
      ${nomeClasse}Model ${atributoClasse}Model = new ${nomeClasse}Model();
      #foreach($column in $tableModel.getColumns())#montaResultSet($column)#end
      return ${atributoClasse}Model;
   }
   private String getOperatorSelect( String operator ) {
      if ( operator.equalsIgnoreCase( "Igual" ) ) { return " = ?"; }
      if ( operator.equalsIgnoreCase( "Diferente" ) ) { return " <> ?"; }
      if ( operator.equalsIgnoreCase( "Maior" ) ) { return " > ?"; }
      if ( operator.equalsIgnoreCase( "MaiorIgual" ) ) { return " >= ?"; }
      if ( operator.equalsIgnoreCase( "Menor" ) ) { return " < ?"; }
      if ( operator.equalsIgnoreCase( "MenorIgual" ) ) { return " <= ?"; }
      if ( operator.equalsIgnoreCase( "Contendo" ) ) { return " LIKE '%?%'"; }
      if ( operator.equalsIgnoreCase( "Iniciando" ) ) { return " LIKE '%?'"; }
      if ( operator.equalsIgnoreCase( "Terminando" ) ) { return " LIKE '?%'"; }
      return null;
   }
}
```

Quadro 32 – *Template* para as classes DAO

APÊNDICE 6 – *Template* para a classe de conexão com o banco de dados

```
public class Conexao {

   private static Connection conn;
   private static Conexao conexao;

   public Conexao() throws InstantiationException, IllegalAccessException,
                           ClassNotFoundException, SQLException {
      String db = "$conn.getDb()";
      String server = "$conn.getServer()";
      String database = "$conn.getDatabase()";
      String user = "$conn.getUser()";
      String password = "$conn.getPassword()";
      if ( password == null ) { password = "''"; }
      Class.forName( getDriver( db ) ).newInstance();
      String url = mountURL( db, server, database );
      conn = DriverManager.getConnection( url, user, password );
   }
   private static String mountURL( String db, String server, String database ) {
      if ( db.equals( "ORACLE" ) )
         return "jdbc:oracle:thin:@" + server + ":1521:" + database;
      if ( db.equals( "SQLSERVER" ) )
         return "jdbc:odbc:" + server;
      return "jdbc:mysql://" + server + ":3306/" + database;
   }
   private static String getDriver( String db ) {
      if ( db.equalsIgnoreCase( "ORACLE" ) )
         return "oracle.jdbc.driver.OracleDriver";
      if ( db.equalsIgnoreCase( "SQLSERVER" ) )
         return "sun.jdbc.odbc.JdbcOdbcDriver";
      return "com.mysql.jdbc.Driver";
   }
   public ResultSet openSQL( PreparedStatement pstmt ) throws SQLException {
      return pstmt.executeQuery();
   }
   public static Conexao getInstanceOf() {
      if ( conexao == null ) {
         try {
            conexao = new Conexao();
            return conexao;
         } catch (Exception e) {
              e.printStackTrace();
              return null;
         }
      }
      else
         return conexao;
   }
   public PreparedStatement setPreparedStatement( String sql ) throws
                                                                SQLException {
      return conn.prepareStatement( sql );
   }
   public void execSQL( PreparedStatement pstmt ) throws SQLException {
      try {
         pstmt.execute();
      } finally {
           pstmt.close();
      }
   }
}
```

Quadro 33 – *Template* para a classe de conexão com o banco de dados

APÊNDICE 7 – *Template* para as classes de controle

```
#set($nomeClasse = ${convert.convertUpperFirst($tableModel.getName())})
#set($atributoClasse = ${tableModel.getName().toLowerCase()})
#set($tabela = ${tableModel.getName()})
#if($typeGenerate ==
"report")#set($nomeClasseGer="${nomeClasse}ControllerReport")
#else#set($nomeClasseGer = "${nomeClasse}Controller")#end
#macro(parametros)#set($count = 0)
#foreach($column in $tableModel.getColumns())#set($count = $count + 1)
#set($type = ${convert.convertType($column.getType(), $column.getPrecision())})
#set($columnName = $column.getName().toLowerCase())
#if($count == ${tableModel.getColumns().size()})$type $columnName
#else$type $columnName,#end#end#end
#macro(valores)#set($count = 0)
#foreach($column in $tableModel.getColumns())#set($count = $count + 1)
#set($type = ${convert.convertType($column.getType(), $column.getPrecision())})
#set($columnName = $column.getName().toLowerCase())
#if($count == ${tableModel.getColumns().size()})$columnName
#else$columnName,#end#end#end
#macro(parametrosDel)#set($count = 0)
#foreach($column in $tableModel.getPk().getColumns())#set($count = $count + 1)
#set($columnName = $column.getName().toLowerCase())
#set($type = $convert.convertType($column.getType(), $column.getPrecision()))
#if($count == $tableModel.getPk().getColumns().size())$type $columnName
#else$type $columnName,#end#end#end
#macro(setParametros $column)
#set($columnUpper = ${convert.convertUpperFirst($column.getName())})
#set($columnLower = ${column.getName().toLowerCase()})
${atributoClasse}Model.set${columnUpper}($columnLower);#end

public class $nomeClasseGer {

   private ${nomeClasse}DAO ${atributoClasse}DAO = null;

   public $nomeClasseGer( ${nomeClasse}DAO ${atributoClasse}DAO ) {
      this.${atributoClasse}DAO = ${atributoClasse}DAO;
   }

   public void inserir( #parametros() ) throws SQLException {
      this.${atributoClasse}DAO.insert(this.new${nomeClasse}(#valores()));
   }

   public void alterar( #parametros() ) throws SQLException {
      this.${atributoClasse}DAO.update(this.new${nomeClasse}(#valores()));
   }

   public void excluir( #parametrosDel() ) throws SQLException {
      ${nomeClasse}Model ${atributoClasse}Model = new ${nomeClasse}Model();
      #set($count = 0)
      #foreach($column in $tableModel.getPk().getColumns())#set($count=$count+1)
      #setParametros($column)#end
      this.${atributoClasse}DAO.delete(${atributoClasse}Model);
   }

   public List getObjects() throws SQLException {
      return this.${atributoClasse}DAO.select();
   }

   public List getObjects(List types, List params, List values, List operators)
                                                          throws SQLException {
      return this.${atributoClasse}DAO.select(types, params, values, operators);
   }
```

```
   public ${nomeClasse}DAO get${nomeClasse}DAO() {
      return this.${atributoClasse}DAO;
   }

   private ${nomeClasse}Model new${nomeClasse}( #parametros() ) {
      ${nomeClasse}Model ${atributoClasse}Model = new ${nomeClasse}Model();
      #foreach($column in $tableModel.getColumns())
      #setParametros($column)#end
      return ${atributoClasse}Model;
   }
}
```

Quadro 34 – *Template* para as classes de controle

APÊNDICE 8 – *Template* para os *servlets*

```
#set($nomeClasse=${convert.convertUpperFirst($tableModel.getName())})
#set($atributoClasse=${tableModel.getName().toLowerCase()})
#set($tabela=${tableModel.getName()})
#if($typeGenerate=="report")#set($nomeClasseGer="${nomeClasse}ServletReport")
#else#set($nomeClasseGer="${nomeClasse}Servlet")#end
#macro(parametros)#set($count=0)
#foreach($column in $tableModel.getColumns())#set($count=$count+1)
#set($type=${convert.convertType($column.getType(),$column.getPrecision())})
#set($columnName=$column.getName().toLowerCase())
#if($count==${tableModel.getColumns().size()})$columnName
#else$columnName,#end#end#end
#macro(parametrosDel)#set($count = 0)
#foreach($column in $tableModel.getPk().getColumns())#set($count=$count+1)
#set($columnName=$column.getName().toLowerCase())
#set($type=$convert.convertType($column.getType(),$column.getPrecision()))
#if($count==$tableModel.getPk().getColumns().size())$columnName
#else$columnName,#end#end#end
#macro(obtemRequest $column)
#set($type=$convert.convertType($column.getType(),$column.getPrecision()))
#set($c1=$convert.convertUpperFirst($column.getName()))
#set($c2=$column.getName().toLowerCase())
#if($type=="String")${type} ${c2}=request.getParameter("${c2}");#end
#if($type=="long")${type} ${c2}=Long.parseLong(util.formatToParseLong(
request.getParameter("${c2}")));#end
#if($type=="double")${type} ${c2} = Double.parseDouble(util.alterPointAndComma(
request.getParameter("${c2}")));#end
#if($type=="java.sql.Date")#set($flag=1)
java.text.SimpleDateFormat ${c2}Format = new
java.text.SimpleDateFormat("dd/MM/yyyy");
${type} ${c2}=new java.sql.Date(((java.util.Date)${c2}Format.parse(
request.getParameter("${c2}"))).getTime());#end#end
#macro(insereFiltro $filter)
#set($colunaLower=$filter.getColumn().getName().toLowerCase())
#set($type=$convert.convertType($filter.getColumn().getType(),$filter.
getColumn().getPrecision()))String ${colunaLower}_input = request.getParameter(
"${colunaLower}_input");
if(!${colunaLower}_input.equals("") ) {
types.add("$type");
params.add("$filter.getColumn().getName()");
#if($type=="char")values.add(${colunaLower}_input);#end
#if($type=="String")values.add(${colunaLower}_input);#end
#if($type=="long")values.add(util.formatToParseLong(${colunaLower}_input));#end
#if($type=="double")values.add(util.alterPointAndComma(${colunaLower}_input));
#end
#if($type=="java.sql.Date")values.add(util.formatToDate(${colunaLower}_input));
#endoperators.add( request.getParameter( "${colunaLower}_option" ) );}#end
#if($typeGenerate == "form")import model.*;#end
public class $nomeClasseGer extends HttpServlet {
   private static final long serialVersionUID = 1L;
   private ${nomeClasse}Controller ${atributoClasse}Controller = null;

   public void init() throws ServletException {
      super.init();
      try {
        this.${atributoClasse}Controller = new ${nomeClasse}Controller(new
                                           ${nomeClasse}DAO());
      }
      catch ( Exception e ) {
         System.out.println( e.getMessage() );
      }
   }
```

```
protected void doGet( HttpServletRequest request, HttpServletResponse
    response ) throws ServletException, IOException {
   this.doPost( request, response );
}
protected void doPost( HttpServletRequest request, HttpServletResponse
          response ) throws ServletException, IOException {
   String action = request.getParameter( "action" );
   Util util = new Util();
   #if($typeGenerate == "form")
   if ( action != null ) {
      String url = "/${nomeClasse}Index.jsp?colorGrid1="+util.getCorGrid(0)+
                "&colorGrid2="+util.getCorGrid(1);
      if ( action.equalsIgnoreCase( "actionIns" ) ) {
         url = "/${nomeClasse}Register.jsp?";
         this.redirect(request, response, url, util);
      }
      if ( action.equalsIgnoreCase( "actionUpd" ) )
         this.${atributoClasse}Update( util, request, response );
      if ( action.equalsIgnoreCase( "actionDel" ) ) {
         this.${atributoClasse}Delete( util, request, response );
         this.redirect(request, response, url, util);
      }
      if ( action.equalsIgnoreCase( "pesquisar" ) ) {
         url = "/${nomeClasse}Search.jsp?colorGrid1="+
               util.getCorGrid( 0 )+"&colorGrid2="+util.getCorGrid(1);
         this.redirect( request, response, url, util );
      }
      if ( action.equalsIgnoreCase( "index" ) )
         this.redirect(request, response, url, util);
      if ( action.equalsIgnoreCase( "inserir" ) ) {
         this.inserir( util, request, response );
         this.redirect(request, response, url, util);
      }
      if ( action.equalsIgnoreCase( "alterar" ) ) {
         this.alterar( util, request, response );
         this.redirect(request, response, url, util);
      }
   }#else
   if ( action != null ) {
      if ( action.equalsIgnoreCase( "report" ) ) {
         String url = "/${nomeClasse}ExecReport.jsp";
         this.redirect(request, response, url, util);
      }
      if ( action.equalsIgnoreCase( "executar" ) )
         this.${atributoClasse}Search( util, request, response );
    }#end
    else {
        RequestDispatcher reqDisp = getServletContext().getRequestDispatcher(
               "/erro.jsp?erro=Página não encontrada!" );
        reqDisp.forward( request, response );
    }
 }
 #if($typeGenerate == "form") #set($flag = 0)
 private void ${atributoClasse}Update( Util util, HttpServletRequest request,
     HttpServletResponse response ) throws ServletException, IOException {
    String url = "/${nomeClasse}Register.jsp?";
    try {
     #foreach($column in $tableModel.getPk().getColumns())
     #obtemRequest($column)#end
     ${nomeClasse}Model ${atributoClasse}Model = this.${atributoClasse}
     Controller.get${nomeClasse}DAO().get${nomeClasse}Model(#parametrosDel());
     request.setAttribute( "${atributoClasse}Model", ${atributoClasse}Model );
     RequestDispatcher reqDisp=getServletContext().getRequestDispatcher(url);
     reqDisp.forward( request, response );
    } catch (SQLException e) {
       response.sendRedirect( "erro.jsp?erro=" + e.getMessage() ); }
```

```
    #if($flag == 1)
    catch (java.text.ParseException e) {
       response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
    }#end
}#set($flag = 0)
private void ${atributoClasse}Delete( Util util, HttpServletRequest request,
      HttpServletResponse response ) throws ServletException, IOException {
   try {
      #foreach($column in $tableModel.getPk().getColumns())
      #obtemRequest($column)#end
      this.${atributoClasse}Controller.excluir(#parametrosDel());
   } catch (SQLException e) {
        response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
   }
   #if($flag == 1)
   catch (java.text.ParseException e) {
      response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
   }#end
}
private void inserir( Util util, HttpServletRequest request,
      HttpServletResponse response ) throws ServletException, IOException {
   try {
      #foreach($column in $tableModel.getColumns())
      #obtemRequest($column)#end
      this.${atributoClasse}Controller.inserir(#parametros());
   } catch (SQLException e) {
       response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
   }
   #if($flag == 1)
   catch (java.text.ParseException e) {
      response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
   }#end
}
private void alterar( Util util, HttpServletRequest request,
      HttpServletResponse response ) throws ServletException, IOException {
   try {
      #foreach($column in $tableModel.getColumns())
      #obtemRequest($column)#end
      this.${atributoClasse}Controller.alterar(#parametros());
   } catch (SQLException e) {
       response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
   }
   #if($flag == 1)
   catch (java.text.ParseException e) {
      response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
   }#end
}#end
#if($typeGenerate == "report")
private void ${atributoClasse}Search(Util util, HttpServletRequest request,
     HttpServletResponse response ) throws ServletException, IOException {
  try {
    #if($report.getColumnsFilter().size() > 0)
    List<String> types = new ArrayList<String>();
    List<String> params = new ArrayList<String>();
    List<Object> values = new ArrayList<Object>();
    List<String> operators = new ArrayList<String>();
    #foreach($filter in $report.getColumnsFilter()) #set($count = $count + 1)
    #insereFiltro($filter)#end#end
    String url = "/${nomeClasse}ShowReport.jsp?colorGrid1="+util.getCorGrid(0)
                 + "&colorGrid2=" + util.getCorGrid( 1 );
      #if($report.getColumnsFilter().size() > 0)
      List objects = this.${atributoClasse}Controller.getObjects(types,
                     params, values, operators);
    #else List objects = this.${atributoClasse}Controller.getObjects();#end
    request.setAttribute( "objects", objects );
    RequestDispatcher reqDisp = getServletContext().getRequestDispatcher(url);
```

```
        reqDisp.forward( request, response );
      } catch (SQLException e) {
         response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
      }
   }#end
   private void redirect( HttpServletRequest request, HttpServletResponse
        response, String url,Util util ) throws ServletException, IOException {
      try {
         List objects = this.${atributoClasse}Controller.getObjects();
         request.setAttribute( "objects", objects );
         RequestDispatcher reqDisp=getServletContext().getRequestDispatcher(url);
         reqDisp.forward( request, response );
      } catch (SQLException e) {
          response.sendRedirect( "erro.jsp?erro=" + e.getMessage() );
      }
   }
}
```

Quadro 35 – *Template* para os servlets

APÊNDICE 9 – *Template* para o arquivo de configuração dos *servlets*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>
    SysVideo
  </display-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>

  #foreach($form in $forms)
  #set($tabelaUpper=${convert.convertUpperFirst($form.getTableBase().getName())})
  #set($tabelaLower=${form.getTableBase().getName().toLowerCase()})
  <servlet>
  <servlet-name>${tabelaUpper}Servlet</servlet-name>
    <servlet-class>servlet.${tabelaUpper}Servlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>${tabelaUpper}Servlet</servlet-name>
    <url-pattern>/${tabelaLower}</url-pattern>
  </servlet-mapping>
  #end

  #foreach($report in $reports)

#set($tabelaUpper=${convert.convertUpperFirst($report.getTableBase().getName())})
  #set($tabelaLower=${report.getTableBase().getName().toLowerCase()})
  <servlet>
    <servlet-name>${tabelaUpper}ServletReport</servlet-name>
    <servlet-class>servlet.${tabelaUpper}ServletReport</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>${tabelaUpper}ServletReport</servlet-name>
    <url-pattern>/${tabelaLower}_report</url-pattern>
  </servlet-mapping>
  #end

</web-app>
```

Quadro 36 – *Template* para o arquivo de configuração dos *servlets*

APÊNDICE 10 – *Template* para página de menu

```
#macro(criaMenu $group)
<tr>
  <td align="center" class="tituloMenu">::$group.getName().toUpperCase()::</td>
</tr>
#end

#macro(criaItensMenu $form)
#set($tabela = ${form.getTableBase().getName().toLowerCase()})
<tr>
  <td align="center" class="itensMenu"><a href="${tabela}?action=index"
        target="main"> $form.getName()</a></td>
</tr>
#end

#macro(criaLinkRelatorio $report)
#set($tabela = $report.getTableBase().getName().toLowerCase())
<tr><td align="center" class="itensMenu"> <a
href="${tabela}_report?action=report" target="main"> $report.getName()
</a></td></tr>
#end

<html>
  <head>
    <link rel="stylesheet" type="text/css" href="folhaEstilo.css">
    <title>SysVideo</title>
    <script src="scripts.js"></script>
  </head>
  <body class="bodyMenu">
    <table align="center">
      <tr>
        <td align="center"> <a href="main.jsp" target="main">
          <img src="images/mHome.gif" border="0" alt="Página Inicial"></a></td>
      </tr>
    </table>
    <table align="center">
    #foreach($group in $groups)
      #criaMenu($group)
      #foreach($form in $group.getFormsSelecteds())
        #criaItensMenu($form)
      #end
    #end
    #set($size = $reports.size())
    #if($size > 0)
      <tr><td align="center" class="tituloMenu">:: RELATÓRIOS ::</td></tr>
      #foreach($report in $reports)
        #criaLinkRelatorio($report)
      #end
    #end
    </table>
  </body>
</html>
```

Quadro 37 – *Template* para a página de menu

APÊNDICE 11 – *Template* para as páginas de cadastro

```
#set($cifrao = "${")
#set($tabelaUpper = ${form.getTableBase().getName().toUpperCase()})
#set($tabelaLower = ${form.getTableBase().getName().toLowerCase()})
#macro(insereItem $column)
#set($type = $convert.convertType($column.getColumnBase().getType(),
                                  $column.getColumnBase().getPrecision()))
#set($nameForm = "${column.getTableResult().getName().toLowerCase()}")
<tr><td align="right" classe="tdLabelCadastros">$column.getDescription()</td>
#if($type == "char")
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"><input class="inputTextChar"
     type="text" name="$convert.convertLowerAll($column.getColumnBase().
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}"/>
     <img src="images/procurar.gif" border="0" onclick="popup(
          '$nameForm?action=pesquisar',350,600,'$convert.convertLowerAll(
          $column.getTableResult().getName())')"></td>
 <td align="left" class="tdInputCadastros"><input class="inputTextRetorno"
     type="text" name="$convert.convertLowerAll($column.getColumnResult().
          getName())" value="" readonly="readonly"></td>
 #else <td align="left" class="tdInputCadastros"><input class="inputTextChar"
           type="text" name="$convert.convertLowerAll($column.getColumnBase().
                getName())" value="${cifrao}${column.getColumnBase().getName().
                toLowerCase()}}"/></td>#end#end
#if($type == "String")
 #if($column.getColumnBase().getSize() < 5)
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"><input class="inputTextChar"
     type="text" name="$convert.convertLowerAll($column.getColumnBase().
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}"/>
     <img src="images/procurar.gif" border="0" onclick="popup(
          '$nameForm?action=pesquisar',350,600,'$convert.convertLowerAll(
           $column.getTableResult().getName())')"></td>
     <td align="left" class="tdInputCadastros"><input class="inputTextRetorno"
         type="text" name="$convert.convertLowerAll($column.getColumnResult().
              getName())" value="" readonly="readonly" ></td>
 #else <td align="left" class="tdInputCadastros"><input class="inputTextChar"
           type="text" name="$convert.convertLowerAll(
                $column.getColumnBase().getName())"
           value="${cifrao}${column.getColumnBase().
                 getName().toLowerCase()}}"/></td>#end#else
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"><input class="inputText" type="text"
     name="$convert.convertLowerAll($column.getColumnBase().getName())"
     value="${cifrao}${column.getColumnBase().getName().toLowerCase()}}"/>
     <img src="images/procurar.gif" border="0" onclick="popup(
              '$nameForm?action=pesquisar',350,600,'$convert.convertLowerAll(
              $column.getTableResult().getName())')"></td>
     <td align="left" class="tdInputCadastros"><input class="inputTextRetorno"
         type="text" name="$convert.convertLowerAll($column.getColumnResult().
              getName())" value="" readonly="readonly" ></td>
 #else <td align="left" class="tdInputCadastros"><input class="inputText"
           type="text" name="$convert.convertLowerAll($column.getColumnBase().
                getName())" value="${cifrao}${column.getColumnBase().getName().
                toLowerCase()}}"/></td> #end#end#end
#if($type == "long")
 #if($column.getColumnBase().getSize() == 10)
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"><input class="inputTextCodigo"
     type="text" name="$convert.convertLowerAll($column.getColumnBase().
```

```
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}" onkeydown="mascaraInt(this,event)"/>
          <img src="images/procurar.gif" border="0"
     onclick="popup('$nameForm?action=pesquisar',350,600,'
            $convert.convertLowerAll($column.getTableResult().getName())')">
</td>
 <td align="left" class="tdInputCadastros"><input class="inputTextRetorno"
     type="text" name="$convert.convertLowerAll($column.getColumnResult().
          getName())" value="" readonly="readonly" ></td>
 #else <td align="left" class="tdInputCadastros"><input class="inputTextCodigo"
           type="text" name="$convert.convertLowerAll($column.getColumnBase().
           getName())" value="${cifrao}${column.getColumnBase().getName().
           toLowerCase()}}" onkeydown="mascaraInt(this,event)"/></td>#end#end
 #if($column.getColumnBase().getSize() == 11)
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"><input class="inputTextCodigo"
     type="text name="$convert.convertLowerAll($column.getColumnBase().
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}" onkeydown="mascaraInt(this,event)"/>
          <img src="images/procurar.gif" border="0" onclick="popup('$nameForm?
               action=pesquisar',350,600,'$convert.convertLowerAll(
               $column.getTableResult().getName())')"></td>
 <td align="left" class="tdInputCadastros"><input class="inputTextRetorno"
     type="text" name="$convert.convertLowerAll($column.getColumnResult().
          getName())" value="" readonly="readonly" ></td>
 #else <td align="left" class="tdInputCadastros"><input class="inputTextCodigo"
          type="text" name="$convert.convertLowerAll($column.getColumnBase().
               getName())" value="${cifrao}${column.getColumnBase().getName().
               toLowerCase()}}" onkeydown="mascaraInt(this,event)"/></td>#end#end
 #if($column.getColumnBase().getSize() != 11 &&
     $column.getColumnBase().getSize() != 10)
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"> <input class="inputTextCodigo2"
     type="text" name="$convert.convertLowerAll($column.getColumnBase().
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}" onkeydown="mascaraInt(this,event)"/>
          <img src="images/procurar.gif" border="0" onclick="popup('$nameForm?
                    action=pesquisar',350,600,'$convert.convertLowerAll(
                    $column.getTableResult().getName())')"></td>
     <td align="left" class="tdInputCadastros"> <input class="inputTextRetorno"
         type="text" name="$convert.convertLowerAll($column.getColumnResult().
              getName())" value="" readonly="readonly"></td>
 #else <td align="left" class="tdInputCadastros"><input class="inputTextCodigo2"
           type="text" name="$convert.convertLowerAll($column.getColumnBase().
           getName())" value="${cifrao}${column.getColumnBase().getName().
           toLowerCase()}}"onkeydown="mascaraInt(this,event)"/></td>#end#end#end
#if($type == "double")
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"> <input class="inputTextValor"
     type="text" name="$convert.convertLowerAll($column.getColumnBase().
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}" onkeydown="mascaraFloat(this,event)"/>
          <img src="images/procurar.gif" border="0"onclick="popup(
                    '$nameForm?action=pesquisar',350,600,'$convert.
                    convertLowerAll($column.getTableResult().getName())')"></td>
     <td align="left" class="tdInputCadastros"> <input class="inputTextRetorno"
         type="text" name="$convert.convertLowerAll($column.getColumnResult().
              getName())" value="" readonly="readonly" ></td>
 #else <td align="left" class="tdInputCadastros"> <input class="inputTextValor"
           type="text" name="$convert.convertLowerAll($column.getColumnBase().
           getName())" value="${cifrao}${column.getColumnBase().getName().
           toLowerCase()}}" onkeydown="mascaraFloat(this,event)"/></td>#end#end
#if($type == "java.sql.Date")
 #if($column.getColumnResult().getName() != "")
 <td align="left" class="tdInputCadastros"><input class="inputTextData"
     type="text" name="$convert.convertLowerAll($column.getColumnBase().
```

```
          getName())" value="${cifrao}${column.getColumnBase().getName().
          toLowerCase()}}" onkeydown="mascaraData(this,event)"/>
          <img src="images/procurar.gif" border="0"
          onclick="popup('$nameForm?action=pesquisar',350,600,'$convert.
          convertLowerAll($column.getTableResult().getName())')"></td>
 <td align="left" class="tdInputCadastros"> <input class="inputTextRetorno"
     type="text" name="$convert.convertLowerAll($column.getColumnResult().
     getName())" value="" readonly="readonly" > </td>
 #else <td align="left" class="tdInputCadastros"><input class="inputTextData"
           type="text" name="$convert.convertLowerAll($column.getColumnBase().
              getName())" value="${cifrao}${column.getColumnBase().getName().
              toLowerCase()}}" onkeydown="mascaraData(this,event)"/></td>#end#end
  </tr>
#end
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="folhaEstilo.css">
    <script src="scripts.js"></script>
  </head>
  #foreach($column in $form.getTableBase().getColumns())
  #set($tabela = $form.getTableBase().getName().toLowerCase())
  #set($columnName = ${column.getName().toLowerCase()})
  <jstlTag:set var="${columnName}"
   value="${cifrao}${tabela}Model.${columnName}}"/>
  #end
  <jstlTag:set var="action" value="${param.action}"/>
  <jstlTag:if test="${cifrao}action == 'actionIns'}">
    <jstlTag:set var="action" value="inserir"/>
    <jstlTag:set var="status" value="Inserindo Registro"/>
  </jstlTag:if>
  <jstlTag:if test="${cifrao}action == 'actionUpd'}">
    <jstlTag:set var="action" value="alterar"/>
    <jstlTag:set var="status" value="Editando Registro"/>
  </jstlTag:if>
  <body onload="initFocus()">
  <form action="${tabelaLower}" method="post" name="${tabelaLower}">
  <table class="tableForm" align="center">
    <tr><td class="tituloTelas" colspan="6"><b> :: ${tabelaUpper} ::
</b></td></tr>
    <tr><td class="tituloTabela" colspan=6><b>${status}</b></td></tr>
    #foreach($column in $form.getColumnsForm())#insereItem($column)#end
    <tr></tr>
    <tr><td align="right" colspan="6"><input class="inputButton" type="submit"
            value="Confirmar"></td></tr>
  </table>
    <input type="hidden" name="action" value="${action}"></form>
  </body>
</html>
```

Quadro 38 – *Template* para as páginas de cadastro

APÊNDICE 12 – *Template* para as páginas de pesquisa

```
#set($cifrao = "${")
#set($nameClass = $form.getTableBase().getName().toUpperCase())
#macro(criaCabecalhoTabela $column)
<td align="left" class="tituloTabela"><b>$column.getName().toUpperCase()</b></td>
#end
#macro(criaItensTabela $columnTableResult)
#set($typeKey = "$convert.convertType($columnForm.getColumnBase().getType(),
     $columnForm.getColumnBase().getPrecision())")
#set($typeValue = "$convert.convertType($columnForm.getColumnResult().getType(),
     $columnForm.getColumnResult().getPrecision())")
#set($tableResult = $convert.convertUpperFirst(
     $columnForm.getTableResult().getName()))
#set($columnResult=${columnForm.getColumnResult().getName().toLowerCase()})
#set($columnBase=${columnTableResult.getName().toLowerCase()})
#set($keyVar=${columnForm.getColumnBase().getName().toLowerCase()})
#set($valueVar=${columnForm.getColumnResult().getName().toLowerCase()})
#if($typeValue=="String")#set($valueVar="'${cifrao}objects.$valueVar}'")#end
#if($typeValue=="char")#set($valueVar="'${cifrao}objects.$valueVar}'")#end
#if($typeValue=="java.util.Date")
#set($valueVar="'${cifrao}objects.$valueVar}'")#end
#if($typeValue=="long")#set($valueVar="${cifrao}objects.$valueVar}")#end
#if($typeValue=="double")#set($valueVar="${cifrao}objects.$valueVar}")#end
#if($typeKey=="String")#set($keyVar="'${cifrao}objects.$keyVar'}")#end
#if($typeKey=="char")#set($keyVar="'${cifrao}objects.$keyVar'}")#end
#if($typeKey=="java.util.Date")#set($keyVar="'${cifrao}objects.$keyVar'}")#end
#if($typeKey=="long")#set($keyVar="${cifrao}objects.$keyVar}")#end
#if($typeKey=="double")#set($keyVar="${cifrao}objects.$keyVar}")#end
<td align="left"><a href="JavaScript:returnForm(${keyVar},${valueVar});">
    ${cifrao}objects.${columnBase}}</a></td>#end
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<html> <head> <link rel="stylesheet" type="text/css" href="folhaEstilo.css">
<script src="scripts.js"></script>_<script language="JavaScript">
function returnForm(key,value) {
  top.window.opener.document.${form.getName().toLowerCase()}.${
convert.convertLowerAll($columnForm.getColumnBase().getName())}.value=key;
  top.window.opener.document.${form.getName().toLowerCase()}.${
convert.convertLowerAll($columnForm.getColumnResult().getName())}.value=value;
  top.window.opener.document.${form.getName().toLowerCase()}.${
convert.convertLowerAll($columnForm.getColumnBase().getName())}.focus();
  window.close(); } </script>
</head>
<body> <table class="tableForm" width="100%">
  <tr><td class="tituloTelas" colspan="8"><b>:: PESQUISA $nameClass
::</b></td></tr>
  <tr> #foreach($column in $columnForm.getTableResult().getColumns())
       #criaCabecalhoTabela($column)#end </tr>
  <jstlTag:forEach items="${objects}" var="objects" varStatus="linha">
    <jstlTag:if test="${cifrao}linha.index % 2 == 0}">
      <jstlTag:set var="cor" value="${param.colorGrid1}" />
    </jstlTag:if>
    <jstlTag:if test="${cifrao}linha.index % 2 != 0}">
      <jstlTag:set var="cor" value="${param.colorGrid2}" />
    </jstlTag:if>
    <tr bgcolor="${cor}">
      #foreach($columnTableResult in $columnForm.getTableResult().getColumns())
      #criaItensTabela($columnTableResult) #end</tr>
  </jstlTag:forEach>
</table></body></html>
```

Quadro 39 – *Template* para as páginas de pesquisa

APÊNDICE 13 – *Template* para as páginas de visualização das informações

```
#set($tabelaUpper=${form.getTableBase().getName().toUpperCase()})
#set($tabelaLower=${form.getTableBase().getName().toLowerCase()})
#set($tabelaFirstUpper=${convert.convertUpperFirst(
                       $form.getTableBase().getName().toUpperCase())})
#set($cifrao="${")
#macro(criaCabecalhoTabela $column)
<td align="left" class="tituloTabela"><b>$column.getDescription()</b></td>#end
#macro(criaItensTabela $column)
#set($type=$convert.convertType($column.getType(),$column.getPrecision()))
#set($columnName=$column.getName().toLowerCase())
#if($type=="long")<td align="left"><jstlTag:out
value="${cifrao}object.${columnName}}" /></td>
#else#if($type=="double")
<td align="left"><jstlTag:out value="${cifrao}object.${columnName}}"/></td>
#else#if($type=="java.sql.Date")
<td align="left"><jstlTag:out value="${cifrao}object.${columnName}}" /></td>
#else<td align="left"><jstlTag:out
value="${cifrao}object.${columnName}}"/></td>#end#end#end#end
#macro(parametros)#set($count=0)
#foreach($column in
$form.getTableBase().getPk().getColumns())#set($count=$count+1)
#set($type=${convert.convertType($column.getType(), $column.getPrecision())})
#set($columnName=$column.getName().toLowerCase())
#if($count == ${form.getTableBase().getPk().getColumns().size()})
${columnName}=${cifrao}object.${columnName}}"
#else${columnName}=${cifrao}object.${columnName}}&#end#end#end
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@taglib prefix="jstlTag" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
 <head>
  <link rel="stylesheet" type="text/css" href="folhaEstilo.css">
  <script src="scripts.js"></script>
 </head>
 <body>
 <table width="100%">
   <tr align="right"><td align="right"><a href="${tabelaLower}?action=actionIns"
       target="main"> <img src="images/novo.gif" border="0"> Incluir </a></td>
   </tr>
 </table>
 <table class="tableForm" width="100%">
  <tr><td class="tituloTelas" colspan="8"><b>::${tabelaUpper}::</b></td></tr>
  <tr>
   <td class="tituloTabela" width="1%"></td>
    #foreach($column in $form.getColumnsForm())
    #criaCabecalhoTabela($column)
    #end
   <td class="tituloTabela" width="1%"></td>
  </tr>
  <jstlTag:forEach items="${objects}" var="object" varStatus="linha">
    <jstlTag:if test="${cifrao}linha.index % 2 == 0}">
      <jstlTag:set var="cor" value="${param.colorGrid1}" />
    </jstlTag:if>
  <jstlTag:if test="${cifrao}linha.index % 2 != 0}">
    <jstlTag:set var="cor" value="${param.colorGrid2}" />
  </jstlTag:if>
  <tr bgcolor="${cor}">
    <td align="center"><a href="${tabelaLower}?action=actionUpd&#parametros()
       target="main"><img src="images/form.gif" border="0" alt="Editar Registro">
    </a></td>
    #foreach($column in $form.getTableBase().getColumns())
    #criaItensTabela($column)#end
```

```
    <td align="center"><a href="${tabelaLower}?action=actionDel&#parametros()
        target="main"> <img src="images/deleteGrid.gif" border="0"
        alt="Excluir Registro"> </a></td>
  </tr>
  </jstlTag:forEach>
 </table>
 <br><b>Legenda</b>
 <table border="0">
  <tr align="center"><td><img src="images/form.gif"
      border="0">Editar Registro</td>
  </tr>
  <tr align="center"><td><img src="images/deleteGrid.gif"
      border="0">Excluir Registro </td>
  </tr>
 </table>
 </body>
</html>
```

Quadro 40 – *Template* para as páginas de visualização das informações