Ferramenta Rad para Geração de Código em Php

Alexandre Cruz Berg[1], Rodrigo Santos Ferraz[2]

Universidade Luterana do Brasil - ULBRA
[1,] berg@ulbra.br
[2,] rs.ferraz@gmail.com

Resumo

Neste artigo descreve-se o estudo e a construção de uma ferramenta para desenvolvimento rápido de aplicação (RAD), durante o ciclo de vida do desenvolvimento de software, utilizando a geração de código unidirecional. O presente trabalho aborda conceitos, reutilização de classes existentes e consolidadas na comunidade *software livre*, motivação para a sua elaboração e um estudo de viabilidade para validação da ferramenta desenvolvida.
Palavras-Chave*:* Desenvolvimento rápido de aplicação; Geração de código; Ciclo de Vida; Software livre

Abstract

This paper describes the study and the construction tool for rapid application development (RAD), during the cycle of life of the software development, using the unidirectional code generation. This present work approaches concepts, reuse of existing class and consolidated in the free software community, motivation for its elaboration and a feasibility study for validation of the developed tool.
Keywords*:* Rapid application development; Code generation; Life cycle; Free software

1 Introdução

O desenvolvimento de programas de computador obedece algumas etapas obrigatórias, segundo a Engenharia de *software,* o ciclo de vida do *software*. Em certos momentos do processo evolutivo do desenvolvimento, mais especificamente a implementação, é preciso empenhar grande esforço e tempo, não importando as dimensões ou finalidades do sistema a ser desenvolvido.

Pode-se dividir esse ciclo em quatro modelos básicos[14]: o tradicional, prototipação, desenvolvimento rápido de aplicação (*Rad)* e o desenvolvimento pelo usuário final. Todas essas abordagens variam conforme a organização que a utiliza.

Este trabalho apresenta uma ferramenta que permite a geração código fonte em *Hypertext Preprocessor (Php)*[4][11] a partir de um esquema relacional implementado num Sistema Gerenciador de Banco de Dados (Sgbd), automatizando o processo de codificação de programas para a *Web*.

O item 2, descreve propriamente a ferramenta proposta por este trabalho, com uma visão geral sobre a mesma e a exposição detalhada dos seus componentes fundamentais. No item 3, é descrita uma abordagem sobre *templates,* o que são e qual seu propósito dentro e fora da ferramenta, juntamente com a integração dos componentes citada dentro do item 2[12]. Por sua vez o item 4 apresenta um plano de validação dos resultados gerados pela ferramenta, descrevendo como foi testado e quais os pontos levados em consideração e, por fim, o item 5 conclui este artigo relatando possíveis expansões da ferramenta bem como suas limitações.

2 Ferramenta

Para suprir a necessidade de geração da código de forma rápida e confiável, necessita-se de uma ferramenta composta por segmentos estáveis e confiáveis. Uma forma interessante seria projetar uma ferramenta com flexibilidade e compatibilidade, com diversos Sgbd, por exemplo, e dividi-la em partes específicas e integradas harmonicamente. Por este motivo, a ferramenta desenvolvida é um conjunto de módulos.

Utilizou-se esta técnica porque é um princípio básico dentro do conceito do "ciclo de vida software" [13]. A modularização foi aplicada para reduzir a complexidade do processo do desenvolvimento do software. O sistema é

dividido em blocos ou em módulos menores. Isso facilita a depuração, manutenção e futuras expansões.

Algumas bibliotecas utilizadas foram desenvolvidas pelas comunidades de *software* livre[15], possuindo assim, boa estabilidade, consolidação de uso e continuação garantidas. Completando, foram desenvolvidas outras bibliotecas específicas para a ferramenta, com recursos capazes de colher, processar e gerar as saídas esperadas para a finalidade deste trabalho.

Logo, a ferramenta final corresponde a um grupo de componentes agregados e relacionados entre si, capazes de buscar informações, processar e gerar uma saída específica, neste caso o código fonte em *Php*.

Utilizou-se a linguagem *Php* para implementar a ferramenta, pois é uma poderosa linguagem *Web* com código fonte aberto e de fácil integração com diversos bancos de dados. Segundo Converse [3], em 1997 o *Php* era usado em cerca de 50.000 "W*eb-sites"* espalhados pelo mundo. Em 1998 eram 100.000 domínios, em 1999 um milhão e em 2000 já eram dois milhões utilizando-o.

Para auxiliar algumas funcionalidades de interface, utilizou-se os recursos de *JavaScript*. O *JavaSCript* é uma linguagem derivada do C e C++, porém distinta na sua aplicabilidade [7]. O *JavaScript* é utilizado normalmente no lado do cliente, ou seja, no navegador.

O Sgbd utilizado para a ferramenta é o *PostgreSql 8.0,* por ser um Sgbd robusto e amplamente utilizado pela comunidade de *Software* Livre.

2.1 Visão Geral

A Ferramenta divide-se em três importantes partes: acesso aos dados, dicionário de dados e o gerador de código. Esses respectivos componentes, são responsáveis pelo fluxo de execução da ferramenta, como mostra a Figura 1.

A Figura 1 ilustra o funcionamento macro, indicando cada componente e seus relacionamentos. A parte referida à "aplicação", é o código gerado, mais as regras de negócio e o seu banco de dados. O código gerado é o resultado prático da ferramenta.

Parte das regras de negócio também são geradas. Essas regras geradas são básicas. Representam apenas métodos simples de inserção, exclusão, alteração e recuperação de dados. Geram uma base de código única e importante para as possíveis manutenções. Nesse ponto, onde se localizam as regras de negócio, pode-se inserir codificação pertinente a cada problema específico, pois apresenta código legível e de fácil interpretação.

A geração de código ocorre de forma unidirecional, ou seja, ele é gerado e, se re-gerado, é substituído pelo último resultado. Por isso, as alterações feitas por intervenção humana no código criado, serão perdidas caso se repita esse processo.

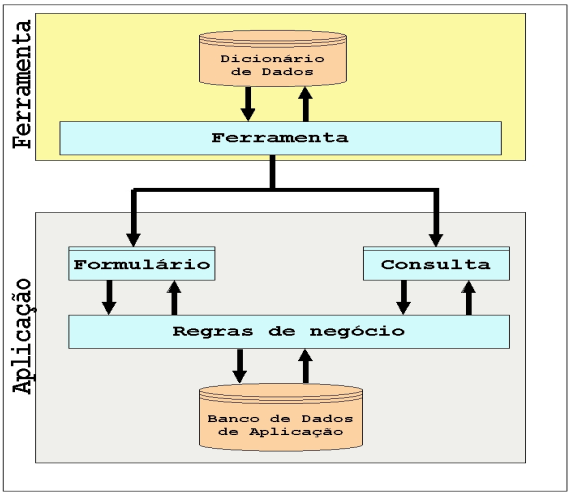

Figura 1 – Visão Geral

O resultado a ser gerado é baseado em algumas regras dentro do esquema relacional da aplicação, logo, as interfaces de *layout* não serão geradas pela ferramenta. Mas o fato de não gerar as telas de interface, não significa que o resultado será apenas código de manipulação de informações. Utiliza-se, neste caso, uma integração com modelos de interface. Os modelos de interface ou de tela de apresentação representam a parte gráfica da aplicação, o *front end*. Assim, além de possuir uma base de telas de apresentação pré-definidas, obtém-se também, a padronização das interfaces. As interfaces geradas são para manipulação de dados (cadastros) e para consultas (navegação). Estas duas interfaces distintas utilizam a mesma camada de regra de negócio, distinguindo-se o seu uso, pelos métodos aplicados a cada uma delas.

2.2 Adodb

O programador engana-se ao desenvolver uma aplicação sem qualquer consideração com o futuro. Não é impossível que os bancos de dados utilizados atualmente, tornem-se inviáveis no futuro e que seja preciso reescrever a aplicação para utilizar outro banco. As funções nativas do *Php* não facilitam este processo caso isso venha a acontecer [2].

A biblioteca *Adodb* é uma classe desenvolvida para a linguagem *Php* que proporciona uma padronização dos comandos para acesso a banco de dados. O *Adodb* permite que sejam realizadas alterações no banco de dados com o mínimo de esforço, usando algumas funções, modificando apenas um parâmetro que define o banco de dados a utilizar [2]. A partir desta mudança, todos os comandos e atributos compatíveis entre os Sgbd continuam funcionando normalmente sem a necessidade de maiores modificações na codificação. A classe *Adodb* também oferece outras abstrações, além das conexões. Este componente possibilita, inclusive, a extração de meta-dados dos Sgbd.

A Figura 2 apresenta a arquitetura macro da classe

Adodb.

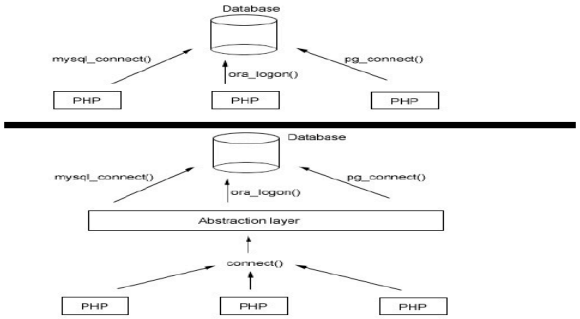

Figura 2 – Arquitetura da classe *Adodb* [8]

É possível visualizar na parte superior da Figura 2, que o *Php* acessando diretamente o banco de dados precisa especializar o código para cada Sgbd a ser utilizado. A parte inferior ilustra a utilização de uma camada de abstração, neste caso a *Adodb*, eliminando a necessidade da aplicação acessar diretamente o banco de dados.

2.3 Smarty

O *Smarty* pode ser considerado um mecanismo interpretador de modelos de interface. Isso significa que é um componente com certo grau de linguagem própria, que a interpretando, gera algum resultado. Esse tipo de sistema de *template* permite a separação da marcação *Html* do código funcional do *Php*, agregando legibilidade e maior organização.

A utilização de modelos de *Layout* é importante porque permite a separação das regras de negócio das telas de apresentação. Permite ainda que se tenha maior controle de padronização de interfaces.

"O *Smarty* utiliza-se de uma macro linguagem que servirá para fazer a substituição de valores. Para isto, uma marcação especial é utilizada e, todo conteúdo existente dentro desta marcação, será analisada pelo *Smarty* e seu conteúdo será substituído pelo respectivo gerado por ele. Possui duas características que o tornam muito potente: a utilização de *caching* e os *plugins"* [6].

2.4 MetaDados

O componente "MetaDados" é responsável por obter todas as informações das tabelas da aplicação e colocá-las em estruturas específicas. Faz uso de recursos da classe *Adodb* para extrair os meta-dados, abstraindo assim, condições específicas de cada Sgbd.É composto por três classes: "MetaDados", "MDChaveEstrangeira" e "MDColuna".

A classe "MetaDados" é a principal, a única que pode ser instanciada dentro da ferramenta. Ela possui métodos para retornar nomes de tabelas, colunas e chaves estrangeiras.

As outras duas classes "MDChaveEstrangeira" e "MDColuna" são apenas estruturas fixas utilizadas para padronizar a exportação destes dados ao chamador, com exceção da classe "MDColuna", que possui um único método, responsável por sinalizar se o campo referenciado é auto-incremento.

Este componente não se comunica com o banco de dados da ferramenta. Conecta-se somente com a base de dados aplicação. Para efetuar esta conexão, recebe um ponteiro para uma conexão pré-estabelecida. Assim, a classe não necessita ter conhecimento ou mesmo buscar em algum lugar os dados necessários à conexão com o banco de dados da aplicação.

2.5 Dicionario

A ferramenta necessita de um repositório de dados específico, capaz de armazenar as informações básicas a serem usadas para a geração de código. O componente "Dicionario" é um conjunto de classes que importa, manipula e retorna informações do dicionário de dados da ferramenta. "Ele pode ser visto como um depósito central que descreve e define o significado de todas as informações utilizadas na construção de um sistema"[10].

As respectivas classes, executam consultas externas, requisitando para o componente "MetaDados", consultas e gravações internas, isto é, utilizando a sua própria base de dados local para manipular as informações. A importação de informações é a sincronização com a base de dados da aplicação a ser gerada. A ela, dizem respeito os meta-dados de tabelas, campos e chaves estrangeiras.

O "Dicionario" é composto por quatro classes: "Tabela", "Campo", "Tipo" e "ChaveEstrangeira".

Outro ponto importante do componente "Dicionario" é a classe "Campo". Esta, por sua vez, possui a responsabilidade de importar e gerenciar os campos que compõem cada tabela. Contém métodos capazes de manipular todas as informações pertinentes a cada campo, como nome, tipo, etc. Para relacionar o tipo de cada campo, utiliza-se a classe "Tipo". Também configura para cada campo o seu tipo de *tag Html.*

O Quadro 1 mostra os tipos de *tag* suportados pela ferramenta. Todos, com exceção dos *tag* "Input Submit" e "Input Reset", são utilizados para campos. Esses dois são utilizados para os formulários.

A classe "Campo*"* ainda determina qual o *tag Html* inicial que será o campo, no momento de sua importação. Quando um campo é inserido, inicialmente recebe o tipo de *tag* "TEXT". Este tipo é o mais comumente usado, pois é uma entrada simples de dados e normalmente texto puro.

A classe "ChaveEstrangeira" é responsável por verificar se um campo é chave estrangeira de alguma tabela. É sua função também inserir as informações pertinentes a uma chave estrangeira, no dicionário de dados. Para determinar se o campo é realmente uma chave estrangeira, utiliza-se do método "GetFK" da classe "MetaDados", a fim de retornar um registro com as devidas informações.

O componente também é o responsável por cadastrar, alterar, excluir e associar funções *JavaScript*. Contém uma classe com métodos necessários para a manipulação destas informações. Nesse cadastro, deve ser indicado um nome para a função, seus parâmetros, uma descrição simples e o "corpo" da mesma.

Quadro 1 – *Tag Html* utilizados na ferramenta

Tipo *Html*	Atributo na Ferramenta	Descrição
Input Text	$TAG_TEXT_DEF	É uma caixa de entrada de dados simples.
Select	$TAG_DROP_DEF	Representa uma caixa de seleção
Input Checkbox	$TAG_CHECK_DEF	É um campo de marcação simples
Input Radio	$TAG_OPTION_DEF	Campo de opção única
Input Hidden	$TAG_HIDDEN_TEXT_DEF	Campo de texto escondido
Input Submit	$TAG_BOTAO_CONFIRMAR	Botão para submeter o formulário
Input Reset	$TAG_BOTAO_LIMPAR	Botão para limpar dados de um formulário

Para associar as funções cadastradas na ferramenta com os campos, deve-se utilizar a interface de configuração do campo. Nesse local, é possível associar uma função para cada evento *Html* pré-definido na ferramenta, ilustrados no Quadro 2.

Quadro 2 – Eventos *Html* pré-definidos na ferramenta

Evento *Html*	Atributo na Ferramenta
OnBlur	$EVENTO_ON_BLUR
OnChange	$EVENTO_ON_CHANGE
OnClick	$EVENTO_ON_CLICK
OnFocus	$EVENTO_ON_FOCUS
OnMouseOut	$EVENTO_ON_MOUSE_OUT
OnMouseOver	$EVENTO_ON_MOUSE_OVER

2.6 Gerador

O componente "Gerador" é a parte que apenas lê as informações gravadas no dicionário de dados e, baseado nelas, gera o resultado final: o código.

Mantém dependência direta das classes do componente "Dicionario", por necessitar de seus respectivos dados. O código gerado divide-se em quatro partes:

- Classe específica: Esta classe gerada é a parte fundamental para o funcionamento da aplicação a ser montada. Contém todos os métodos necessários para a manipulação e buscas de dados na respectiva tabela para a qual foi gerada.
- Arquivo principal: Este é o arquivo chamado diretamente por alguma interface. Na sua inicialização é verificado o tipo de ação que deve tomar. Por último, faz uma chamada para o interpretador *Smarty*, a fim de gerar dinamicamente as informações e instanciar o *template* de apresentação.
- *Templates* de apresentação: É o *front end* gerado. Possui uma estrutura compatível a ser interpretada pelo *Smarty*. Os *templates* gerados são para a tela de exibição e para o formulário de entrada de dados. Esse resultado é agregado a outras características do *template* interno, descrito na sub-seção 3.2.
- Arquivo *JavaScript*: É um arquivo contendo todas as funções de *JavaScript* necessárias ao funcionamento da aplicação. Este, por sua vez, é incluído no *template* de apresentação, pois todos os *scripts* são aplicados à interface.

Cada aplicação é gerada dentro de um diretório com seu respectivo nome, dentro do respectivo, cria-se as seguintes pastas:

- Classes: Onde serão gravadas todas as classes básicas para o funcionamento da aplicação. Dentro desta também são copiados os componentes *Adodb* e *Smarty*.
- Front: Neste local são gravados os arquivos principais, os quais recebem chamadas diretas. Percorrendo mais um nível dentro desta pasta, encontra-se os diretórios "templates" e "templates_c", que armazenam os *templates* de apresentação e os compilados pelo Smarty, respectivamente.
- Js: Contém normalmente um único arquivo com a extensão "*.js",com as funções *JavaScript* geradas.
- Sistema: O diretório "Sistema" concentra todas as classes específicas geradas pela ferramenta.

O Gerador é composto por duas classes: "Gerador" que é a classe principal deste componente. É ela que recebe as solicitações de geração de

código."ParserTemplate" que é uma classe de apoio, onde é utilizada para interpretar e modificar resultados baseado nos *templates* internos.

3 Templates

O desenvolvimento de um sistema *Web* complexo, com funcionalidades diversas e com um layout amigável é incomodo e desorganizado se for feito de maneira trivial.

Para aumentar a produtividade, bem como diminuir a quantidade de código e separá-lo do *layout Html*, é preciso utilizar modelos de implementação.

3.1 Uso de *Templates*

A utilização de *templates* na programação de sistemas *Web*, especificamente em *Php*, tornou-se muito importante, principalmente em sistemas complexos.

Visando aumentar a compatibilidade entre as interfaces, separando completamente o código *Php* do *Html*, o presente trabalho aplica a utilização de *templates* para agregar reuso e abstração das diferentes necessidades dos códigos implementados.

Baseado nesta técnica, utilizou-se de modelos de códigos dentro da própria ferramenta. Assim, é possível aplicar uma estrutura padrão sem mexer diretamente no código da ferramenta, como demonstrado na seção 3.2, deste artigo.

3.2 *Templates* Internos

Os *templates* internos facilitam a geração do código, uma vez que comportam o molde principal do funcionamento do código a ser gerado. Cada *template* possui características próprias e são projetados especificamente para a ferramenta.

São utilizados como modelos de código pelo componente "Gerador". Possui características obrigatórias, como palavras chaves que serão substituídas por trechos de códigos, como ilustra a Figura 3:

```
function {NOME_CLASSE}()
{
       parent::StdFunc(); // instancia a classe
pai
       register_shutdown_function(array($this,
"_destrutor_{NOME_TABELA}"));
}
```

Figura 3 – Exemplo de código de um *template* interno

Na Figura 3, nota-se as palavras chaves delimitadas pelos caracteres "{" e "}". A palavra chave mais os caracteres delimitadores formam um termo a ser alterado pelo gerador de código. Assim, o *template* interno deve conter todas as palavras chaves necessárias ao pleno funcionamento do código gerado.

A ferramenta necessita de dois *templates* para código e um para a apresentação. Os *templates* de código são:

- Template para a classe específica: Este *template* é um molde com as funções necessárias ao funcionamento da classe específica. Contém modelos de funções para adição, exclusão, alteração e recuperação de registros. Possui também uma função especial onde são validados os valores a serem inseridos ou atualizados no banco de dados. Essa validação é baseada no atributo "not null" de cada campo. É neste ponto onde, depois do código gerado, aconselha-se a implementar novos recursos de validação.
- Template para o arquivo principal: Este é um modelo de *Script* que contém o molde para as principais chamadas para a classe específica. Implementa também a instância e parametrização da classe *Smarty*.

Completando os *templates* usados pela ferramenta, existem ainda, os *templates* de apresentação. Estes modelos devem conter um único campo chave, onde será injetado o formulário gerado ou da tabela de apresentação dos dados. Cada tabela a ser gerada deverá ser associada a dois *templates*: Um para o formulário e outro para a tela de exibição de dados. Todos os *templates* disponíveis devem estar gravados no diretório "modelos", dentro dos sub-diretórios "cadastro" e "navegacao". A associação do modelo com a tabela a ser gerada é realizada pelo próprio nome do modelo, eliminando assim, um relacionamento mais complexo dentro do banco de dados.

A Figura 4 ilustra um *template* interno de apresentação, referente a um formulário de entrada de dados.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
<head>
<title>Título do Documento</title>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
</head>

<body>
<table width="100%" height="100%" border="0">
  <tr>
    <td height="70" colspan="3"><img
src="templates/imagens/banner1.jpg" width="800"
height="100"></td>
  </tr>
  <tr>
    <td width="10%" height="320"> </td>
    <td width="68%"
valign="top">{CORPO_CADASTRO}</td>
    <td width="10%"> </td>
  </tr>
  <tr>
    <td colspan="3"> </td>
  </tr>
</table>
</body>
</html>
```

Figura 4 – Exemplo de um *Template* interno de apresentação

Esse arquivo deve conter o *layout* desejado e, obrigatoriamente, o termo "{CORPO_CADASTRO}". Esse termo será substituído pelo formulário de cadastro, gerado pela ferramenta. Assim, pode-se criar a interface, gravar no local indicado e apenas associar à tabela a ser gerada.

Os mesmos requisitos e possibilidades aplicam-se ao *template* para a tela de navegação. A diferença mais importante é a palavra chave, obrigatoriamente contida dentro do modelo. O termo obrigatório, nesse caso, deve ser "{CORPO_NAVEGACAO}". Também aconselha-se a criação de um botão com uma chamada de função para retornar ao histórico anterior, no navegador.

3.3 *Templates* Externos

O resultado final do modelo de apresentação é um *template* externo. Considera-se o mesmo externo porque não faz parte, nem é recurso utilizado pela ferramenta. É um recurso a ser utilizado pela aplicação.

Inicialmente, o "Gerador" lê os modelos selecionados, descritos na seção 3.2, para a tabela a ser gerada e, com base em suas informações, gera um formulário padrão e uma tabela com os respectivos campos a serem exibidos na tela de navegação. O formulário contém os campos configurados pela ferramenta e é inserido no modelo e, enfim, gravado em seu local de destino. O mesmo ocorre para a tela de exibição.

3.4 Integração

A união dos componentes implementados, bem como os reutilizados requer uma padronização. Para que se possa encapsular os diversos métodos, implementados dentro das classes membros dos componentes, cada componente acessa diretamente as suas respectivas tabelas, dentro do dicionário de dados. Com esta regra, pode-se concentrar o acesso ao banco de dados em um único método, para cada tabela. Isso facilita e normatiza os devidos procedimentos.

Os *templates* internos, por possuírem obrigatoriamente uma estrutura conhecida, integram-se facilmente com os componentes, mais precisamente com o "Gerador". É importante que o mesmo possua todas as palavras chaves a serem substituídas pelo "Gerador". A ordem em que encontram-se no *template* é irrelevante, importando apenas o contexto e a sua lógica interna.

A Figura 5 exemplifica um trecho de código de um *template* interno.

```
function {NOME_CLASSE}()
{
        parent::StdFunc(); // instancia a classe
pai
        register_shutdown_function(array($this,
"_destrutor_{NOME_TABELA}"));
}
function Add_{NOME_TABELA}({PARAM_ADD})
{
        $Add_{NOME_TABELA}_RET = false;
        $MontouRegistro = $this-
>_montaRegistro_{NOME_TABELA}({PARAM_ADD});
        if ($MontouRegistro){
            $Add_{NOME_TABELA}_RET =
parent::ExecutaAcaoDML($this->ACAO_INSERT, $this-
>Tabela,           $this->REGISTRO);
        };
      return $Add_{NOME_TABELA}_RET;
}
```

Figura 5 – Trecho de código de um *template* interno possível de alteração

A etapa final, baseada em todas as informações colhidas no esquema relacional da aplicação e com as configurações feitas pelo usuário, é a geração prática do código.

4 Validação

A validação final da ferramenta consiste na elaboração de um estudo de viabilidade para gerar e abordar os resultados. Como a ferramenta foi desenvolvida com o propósito de ganhar tempo de produção, na etapa rotineira do ciclo de vida de um *software*, deu-se maior ênfase a este aspecto, combinando com a qualidade e validade do código gerado.

Para apurar o processo completo, o caso criado foi um sistema de pedidos rápidos. Este caso consiste em

alguns cadastros básicos interligados por um sistema de navegação simples. Os requisitos do sistema, referente a cadastros são: Clientes e Produtos.

A Figura 6 ilustra o diagrama de Entidades-Relacionamento, criado para o estudo de viabilidades.

A tabela *"*tbtipopessoa*"* possui uma estrutura muito simples. Deverá armazenar, normalmente dois registros, contendo as descrições de tipos de pessoa: "Física" e "Jurídica". Necessita de uma tabela específica para armazenar estes valores, mesmo que praticamente imutáveis, pois a ferramenta criará uma "Drop"*,* na tela de apresentação, com os valores contidos na respectiva. Esse campo será apresentado, desta forma, no cadastro de clientes.

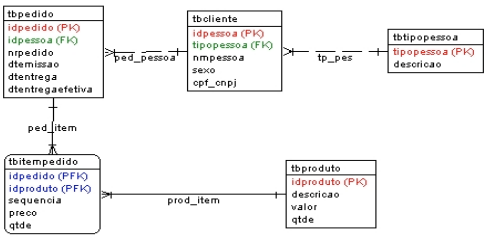

Figura 6 – Diagrama Entidade-Relacionamento para o estudo de viabilidades

O cadastro de clientes compreende em informações relativas ao comprador, requisitante do pedido. Contém informações básicas, como "nome", "cpf/cnpj", "sexo" e "tipo de pessoa" (física ou jurídica). O "tipo de pessoa" é uma chave estrangeira referente ao cadastro de tipos de pessoa.

O cadastro de produtos contém informações sobre o item a ser incluído no pedido. Deve armazenar informações básicas como a descrição do produto, quantidade e valor do mesmo.

No que se refere às regras do negócio, parte do código será gerado pela ferramenta e outra parte necessitará de intervenção humana para refinamentos de programação. As regras de negócio, neste caso aplicado, são as condicionais necessárias para efetuar propriamente um pedido.

Os pedidos compreendem a parte mais dinâmica do sistema. Cada pedido deve possuir obrigatoriamente um cliente, uma dada de emissão, data de entrega e um número de pedido. Pode possuir zero ou "n" itens distintos. Considera-se que o pedido está em aberto quando o mesmo não possuir uma data de entrega efetiva. No momento que o usuário mandar fechar o pedido, o sistema deve colocar a data atual como entrega efetiva, baixando assim, as quantidades de cada produto no seu estoque.

Neste ponto, deve haver intervenção de um programador para implementar as regras necessárias ao pleno funcionamento da presente rotina, uma vez que a ferramenta deve gerar apenas os cadastros básicos. Há também a necessidade de interferência no *layout*, para garantir o pleno funcionamento do sistema. Para fins de aproveitamento de código, utiliza-se a classe específica como base, inserindo nela, todas as implementações necessárias.

Para testar o funcionamento do código gerado, inicialmente inseriu-se valores válidos em todos os campos disponíveis, a fim de validar o funcionamento básico. Com esta etapa validada, pretende-se testar as demais funcionalidades, que são as validações de alguns campos e teste de integração com *JavaScript*.

O teste implica também em gerar o código para três Sgbd: *PostgreSql, Firebird* e *MySql*[9][5].

O comportamento da ferramenta deve manter-se o mesmo para quaisquer das respectivas conexões, devendo mudar apenas aspectos específicos de cada uma, como campos auto-incremento, por exemplo. Para isso, utiliza-se da classe *Adodb*, como descrito na sub-seção 2.2.

O código gerado poderá ser utilizado em qualquer dos três Sgbd escolhidos para a validação, necessitando apenas a alternância do tipo de Sgbd a ser conectado. Desta forma, consegue-se portabilidade da aplicação. Para fins de validação de conexão da ferramenta, bem como a extração de meta-dados específicos de cada Sgbd, são gerados três esquemas relacionais nos devidos Sgbd [1].

4.1 Estudo de Viabilidade

O estudo de viabilidade, demonstrado aqui, divide-se em três partes: Configuração e geração de código do sistema para estudo, dentro da ferramenta, análise do código gerado e teste de uso do sistema final.

Os esquemas relacionais utilizados para validar a ferramenta serão criados em momentos distintos. Cada Sgbd utilizado, irá gerar uma "aplicação" dentro da ferramenta. Isso é necessário para não haver configurações distintas entre eles.

Inicialmente, após criar o esquema relacional dentro do Sgbd, criou-se dentro da ferramenta, uma entrada para a aplicação, chamada de "EstudoDeCaso". Está é a aplicação onde a ferramenta armazenará as informações para geração do código.

O passo seguinte, é a importação da estrutura descritiva das tabelas contidas no esquema relacional cadastrado para a aplicação, dentro da ferramenta. Aqui, utilizou-se o recurso da ferramenta onde pode-se importar todas as tabelas contidas no banco de dados da aplicação.

Após as informações das tabelas estarem inseridas no dicionário de dados da ferramenta, inicia-se a importação dos seus respectivos campos, juntamente com seus atributos. Neste ponto, necessita-se de uma intervenção unitária, ou seja, importar os campos para cada tabela. Com os campos gravados dentro do dicionário,

pode-se iniciar as devidas configurações, como as descrições dos referidos campos ou a otimização dos atributos quando o campo for do tipo "Drop*"*.

Após todos os campos estarem devidamente configurados, deve-se gerar as classes básicas. Essas classes são utilizadas para especificar o acesso ao banco de dados, contendo as devidas configurações, a classe de abstração de dados e o componente de interpretação de *templates*.

Após a geração individual de cada tabela, obtém-se o código final, gerado pela ferramenta, como descrito na seção 2.6. A geração do código acontece de forma instantânea e, somando com o tempo de configuração, obtém-se um grande ganho de tempo. O resultado gerado é um código de qualidade e com ótima clareza lógica.

A Figura 7 exemplifica um método gerado pela ferramenta. Comparando, por exemplo, as Figuras 5 e 7, nota-se que claramente que o código presente na Figura 5, é a base para o código apresentado pela Figura 7. Inseriu-se a parte dinâmica dentro do modelo de código interno, resultando em uma saída funcional e coerente.

Nota-se também, que os comentários inseridos no código também sofrem alterações. Isso ocorre porque o *template* interno permite comentários, podendo ainda, inserir palavras chaves a fim de personalizar os mesmos.

Todos os métodos foram perfeitamente gerados, sem quaisquer erros inesperados. As informações contidas no dicionário de dados da ferramenta são suficientes para atender ao propósito do trabalho, resultando em uma base consistente.

```
// Função responsável por inserir registros na
tabela tbpedido
// Início de Add_tbpedido
function Add_tbpedido($dtemissao, $dtentrega,
$dtentregaefetiva, $idpessoa, $nrpedido)
{
        $Add_tbpedido_RET = false;
        $MontouRegistro = $this-
>_montaRegistro_tbpedido($dtemissao, $dtentrega,
$dtentregaefetiva, $idpessoa, $nrpedido);
        if ($MontouRegistro){
        $Add_tbpedido_RET =
parent::ExecutaAcaoDML($this->ACAO_INSERT, $this-
>Tabela, $this->REGISTRO);
         };// Fim do if ($MontouRegistro)
         return $Add_tbpedido_RET;
}
// FINAL de Add_tbpedido
```

Figura 7 – Exemplo de código gerado

A Figura 8 demonstra um método para validação de campos obrigatórios. Todos os campos gerados na classe específica e, com o atributo "*not null*" no esquema relacional da aplicação, são validados dentro deste método.

```
function _validaCamposNotNull($descricao)
{
        // Início da validação do campo descricao
        if ($descricao == ''){
            return false;
         };
         // Fim da validação do campo descricao

         return true;
}
```

Figura 8 – Exemplo do método de validação de campos obrigatórios

Os formulários gerados não necessitam de alteração na sua parte lógica ou estética, podendo ser utilizados diretamente. Isso agrega imenso valor à ferramenta, pois é capaz de gerar código integrado com o motor de *templates*, completamente funcional, em pouco tempo.

As telas de navegação, onde pode-se listar os registros de uma tabela e chamar os formulários para manipulação, também apresentam um resultado satisfatório ao propósito deste trabalho. O código gerado não necessita alteração. Permite listar todos os elementos de uma tabela. A intervenção necessária, neste ponto, é o ajuste visual do *layout*. Isso porque o gerador apenas aloca cada campo em uma célula de uma tabela *Html*. Com as alterações realizadas, os testes de usabilidade mostraram-se positivos. Há precisão nas informações manipuladas e as validações garantem a consistência dos dados.

A respeito das telas de apresentação, pode-se afirmar que na maioria dos casos não necessita de programação para ser utilizada. Em alguns casos mais complexos, onde efetua consultas sob uma tabela com muitos registros, poderá haver uma certa demora na atualização da tela, no navegador. Nesse caso, será necessário programação para limitar essa consulta, aplicando filtros ou mesmo técnicas de paginação.

Os formulários de cadastro operam de forma estável sem necessidade de programação quando baseados em uma única tabela ou seja, para cadastros simples. Sempre que houver necessidade de uma tela de cadastro mais complexa, por exemplo *mestre-detalhe,* onde um registro pai lista e insere valores de itens, haverá programação após a geração do código pela ferramenta.

Nos demais casos, onde for um formulário simples, a ferramenta mostrou-se funcional e cumpriu seu objetivo principal de ganhar tempo na etapa rotineira de programação.

As validações de acesso à Sgbd pela ferramenta foram plenas e positivas. A ferramenta mostrou-se segura e funcional para os três Sgbd, cumprindo uma de suas finalidades: ser multi-banco. Deve-se isso, principalmente, aos componente *Adodb* e "MetaDados". Os respectivos componentes, combinados, possibilitam executar conexões e extrações de informações de diferentes Sgbd.

Por fim, os testes executados nos diferentes Sgbd, por parte da aplicação, mostraram-se positivos. Para todos, a execução foi plena e satisfatória, não apresentando diferenças na sua execução.

5 Conclusão

A geração rápida de código é um fator determinante à variável "tempo", em um projeto de desenvolvimento de *software*. A ferramenta desenvolvida atinge seu objetivo gerando um código de qualidade e confiável em pouco tempo, se comparado aos métodos tradicionais, a diferença é imensa.

Quando, em um projeto, tem-se um planejamento bem definido, com um modelo E-R (entidade-relacionamento) consistente, a ferramenta torna-se muito importante para implementar as etapas rotineiras de programação.

A ferramenta apresenta uma interface simples, de fácil utilização e navegabilidade. Mesmo assim, a sua utilização direciona-se ao pessoal capacitado para o desenvolvimento na linguagem *Php*, visto que o objetivo deste trabalho é auxiliar no desenvolvimento rápido de aplicações e não um sistema para usuário final. A ferramenta desenvolvida, por utilizar programação modularizada e reaproveitar componente de terceiros, mais especificamente *software livre*, possui grande flexibilidade de manutenção, facilitando assim, correções de erros e futuras expansões.

Apresenta também boa portabilidade por ser desenvolvido na linguagem *Php* e com base de dados em *PostgreSql*. Por tratar-se de uma aplicação *Web*, possui requisitos mínimos para a estação cliente, necessitando somente de um computador com navegador *Microsoft Internet Explorer 5.0* ou superior, ou o *Mozilla Firefox 1.7* ou superior.

Existe a possibilidade de expandir as funcionalidades desta ferramenta. Em um primeiro momento, sugere-se ampliar a quantidade de informações armazenadas no *dicionário de dados* para possibilitar mais opções na configuração do código a ser gerado. Pode-se ainda melhorar a integração com os modelos *Html*, possibilitando, por exemplo, a configuração do tamanho das células.

Uma nova funcionalidade interessante que pode ser implementada, é a validação para os tipos de dado de cada campo dentro do código *Php*. Pode-se validar se deve ser, obrigatoriamente, um número ou "string".

Enfim, as expansões desta ferramenta são limitadas às necessidades de desenvolvimento, o que deixa uma eterna lacuna sobre futuras melhorias. Essas mudanças poderão ser constantes, adaptando-se aos novos paradigmas.

Dentre as limitações deste trabalho, pode ser distinguida, como a mais relevante, a obtenção das informações de chaves estrangeiras dos Sgbd. Isso ocorre porque a classe de abstração *Adodb* não oferece este suporte, obrigando o componente "MetaDados" a efetuar as consultas específicas para cada Sgbd.

Referências

[1] ANDRADE, A. D. *PHP Nuke - Integração, Administração e Desenvolvimento*. São Paulo: Visual Books, 2004. 146p.

[2] CÉSAR, P. *Uma introdução à classe ADODB*. Disponível por WWW em: <http://www.vivaolinux.com.br/artigos/verArtigo.php?codigo=634>.Consultado em: Junho de 2006.

[3] CONVERSE, T, PARK, J., MORGAN, C. *PHP5 and MySql Bible*. Indiana: Wiley Publishing, Inc, 2004.1042p.

[4] DOSOOYE, N. PHPCODEGENIE Dados do projeto. Disponível em: <http://phpcodegenie.sourceforge.net>. Consultado em Maio de 2006.

[5] Firebird: Funcionalidades. Disponível em: <http://www.firebase.com.br/cgi-bin/firebase.cgi/artigo?ID=154>. Consultado em: Junho de 2006.

[6] FLOROIU, C. Zend Company. Disponível em: <http://www.zend.com/zend/tut/tutorial-cezar.php>. Consultado em: Junho de 2006.

[7] GOODMAN, D. *JavaScript Bible*, 4th Edition. New York: Hungry Minds, Inc., 2001. 1327p.

[8] MELONFIRE, I. *PHP – Application Development With ADODB*. Disponível em: <http://www.linhadecodigo.com.br/artigos.asp?id_ac=102&sub=0>. Consultado em: Junho de 2006.

[9] MySql: Documentação. Disponível em: <http://dev.mysql.com/doc/refman/4.1/pt/index.html> Consultado em: Junho de 2006.

[10] OLIVEIRA, J. V. de. Dicionário de Dados. Disponível por WWW em <http://w3.ualg.pt/~jvolivei/ep/dd.pdf>. Consultado em Junho de 2006.

[11] PHP-Nuke: O que é. Disponível em: <http://www.nukebrasil.org>. Consultado em: Maio de 2006.

[12] ScriptCase: Institucional. Disponível em: <htttp://www.scriptcase.com.br/site/produtos>. Consultado em: Maio de 2006.

[13] SIBLEY, E. ; MANTEI, M. ; TEOREY, T. J.. *Cost/Benefit analysis for incorporating human factors in the software lifecycle*. Commu-nications of ACM: 2000

[14] STAIR, R. M. ***Princípios de Sistemas de Informação: uma Abordagem Gerencial***. São Paulo: LTC, 1999. 495p.

[15] TAURION, C. *Software Livre. Potencialidades e Modelos de Negócio*. Rio de Janeiro: Brasport, 2004.

Alexandre Cruz Berg é doutor em Informática pela Universidade de Ilhas Baleares na Espanha, onde defendeu sua tese em fevereiro de 2006. É graduado em Informática pela PUCRS, onde recebeu o título em julho de 1991. Possui especialização em Ciência da Computação pela UFRGS, onde recebeu o título em novembro de 1996. É professor da Ulbra nas disciplinas de Algoritmos e Programação, Interface Homem Computador e Computação Gráfica. Faz parte do grupo de pesquisa de Processamento de imagens da Ulbra.

Rodrigo Santos Ferraz é graduado em Sistemas de Informações pela Ulbra, onde recebeu o título em dezembro de 2006. Atua em desenvolvimento de sistemas de controle empresariais.