Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

ALISSON RODRIGUES DE SOUSA

# APLICAÇÃO DE PADRÕES DE PROJETO COM A LINGUAGEM PHP

Belo Horizonte
2009

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Análise de Sistemas

# APLICAÇÃO DE PADRÕES DE PROJETO COM A LINGUAGEM PHP

por

ALISSON RODRIGUES DE SOUSA

Monografia de Final de Curso

Prof. Ângelo de Moura Guimarães

Belo Horizonte
2009

ALISSON RODRIGUES DE SOUSA

# APLICAÇÃO DE PADRÕES DE PROJETO COM A LINGUAGEM PHP

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Especialista em Informática.

Área de concentração: Análise de Sistemas.

Orientador: Prof. Ângelo de Moura Guimarães

Belo Horizonte
2009

Sousa, Alisson Rodrigues de.

Aplicação de padrões de projeto com a linguagem PHP [manuscrito] / Alisson Rodrigues de Sousa. – 2009.

vi, 39 f., enc. : il.

Orientador: Ângelo de Moura Guimarães.

Monografia (especialização ) – Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

1. Computação – Teses. 2. Analise de Sistemas– Teses. . I. Guimarães, Angelo de Moura II Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. III. Título.

## AGRADECIMENTOS

Agradeço à todo o corpo docente do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais pela dedicação empenhada na formação de profissionais qualificados para o mercado de trabalho e em especial ao professor Ângelo de Moura Guimarães, pela orientação deste trabalho.

Agradeço também à minha namorada Lívia pela paciência e apoio durante os 18 meses de idas e vindas do interior de minas para sua capital, com o objetivo de alcançar o sonho de ser um profissional especializado em ciência da computação pela Universidade Federal de Minas Gerais.

## RESUMO

O principal objetivo deste trabalho é apresentar de forma clara e concisa a aplicação de padrões de projeto de software com a linguagem PHP. Durante o trabalho são apresentados os principais conceitos que envolvem a orientação a objetos e sua aplicação na linguagem PHP. É apresentado também um breve histórico da linguagem PHP, mostrando sua evolução rumo ao suporte à orientação a objetos, suporte este que passa a ser bem completo na versão 5 da linguagem. Durante o desenvolvimento do trabalho são apresentados exemplos de cada categoria dos padrões de projeto, sendo eles o padrão *Singleton*, *Method Factory* e *Abstract Factory* entre os padrões de criação, *Composite, Decorator* e *Façade* entre os padrões Estruturais e *Iterator, Observer* e *Template Method* como representantes dos padrões Comportamentais. Os padrões são utilizados em conjunto para a formação de uma aplicação exemplo.



**Palavras-chave:** Padrões de Projeto de Software, Linguagem PHP, Programação Orientada a Objetos, Desenvolvimento para WEB.

## ABSTRACT

The main objective of this work is to present a clear and concise application of software design patterns with PHP language. During the work are the main concepts involving the orientation to objects and its application in the PHP language. It also presented a brief history of the PHP language, showing its evolution to support the orientation towards the object, that this support will be complete well in version 5 of the language. The work also presents three examples of each category of design patterns, which were the Singleton pattern, Factory Method and Abstract Factory patterns between creation, Composite, Decorator and Façade patterns between structural and Iterator, Observer and Template Method as representatives of Behavior patterns. The patterns are used together to form an application example.



**Keywords:** Software Design Patterns, PHP language, Object Oriented Programming, Web Development.

**LISTA DE FIGURAS**

## LISTA DE SIGLAS

| | |
|---|---|
| ASP | *Active Server Pages* |
| CLR | *Common Language Runtime* |
| GNU GPL | *General Public License* |
| GoF | *Gang of Four* |
| HTML | *HyperText Markup Language* |
| HTTP | *Hypertext Transfer Protocol* |
| IIS | *Internet Information Services* |
| JSP | *Java Server Page* |
| MSIL | *Microsoft Intermediate Language* |
| PHP | *Hypertext Preprocessor* |
| POO | Programação Orientada a Objetos |
| SOAP | *Simple Object Access Protocol* |
| SQL | *Structured Query Language* |
| UML | Linguagem de Modelagem Unificada |
| WEB | *World Wide Web* |
| XML | *Extensible Markup Language* |

**SUMÁRIO**

# 1. INTRODUÇÃO

Em qualquer atividade desempenhada pelas pessoas, seja em suas atividades domésticas ou profissionais, há sempre problemas que se repetem. Quando esta recorrência acontece, é natural que se procure formular uma solução e aprimorá-la com o tempo, mas sempre repetindo sua base. Esta é a idéia dos Padrões de Projeto, termo que segundo a Wikipédia (2009), foi criado na década de 70 pelo arquiteto Christopher Alexander para designar soluções genéricas, documentadas e associadas a um problema, de forma que qualquer pessoa possa aplicá-las apenas adaptando-as ao seu contexto corrente.

Os padrões de projeto assumem a posição do senso comum, onde é consenso que é perda de tempo “reinventar a roda” e permite um melhor aproveitamento do tempo dos desenvolvedores. Com a possibilidade de se utilizar soluções testadas e comprovadas para problemas que aparecem durante o desenvolvimento, os desenvolvedores terão mais tempo para se dedicar às outras etapas do desenvolvimento.

A linguagem PHP, uma das mais utilizadas para o desenvolvimento WEB, passou , segundo Zandstra (2006), na sua 5º versão a suportar em sua essência as potencialidades do desenvolvimento orientado a objetos. Esta evolução permite que se desenvolvam aplicações mais coesas e mais extensíveis, motivando ainda mais o seu uso.

Este trabalho pretende abordar a aplicação de alguns padrões de projeto com a linguagem PHP. Serão apresentados os padrões de criação *Singleton*, *Factory Method* e *Abstract Factory*, os padrões estruturais *Composite*, *Decorator* e *Façade* e os padrões Comportamentais *Iterator*, *Observer* e *Template Method*. Para exemplificar a utilização dos padrões de projeto com o PHP, Foi criada uma aplicação exemplo utilizando estes padrões de projeto. As implementações dos padrões foram criadas a partir do estudo de toda a bibliografia apresentada neste trabalho, podendo alguns exemplos terem alguma similaridade com algum exemplo existente nesta documentação.

## 1.1. Objetivos

Nesta sessão são apresentados o objetivo geral e os objetivos específicos deste trabalho.

### 1.1.1. Objetivo Geral

Apresentar de forma clara os conceitos que envolvem os padrões de projeto e sua aplicação com a linguagem PHP, que apesar de não ser uma linguagem orientada a objetos já suporta em sua essência os conceitos deste paradigma e permite extrair todas as vantagens de sua utilização.

### 1.1.2. Objetivos Específicos

Os objetivos específicos são:

- Pesquisar e apresentar fundamentação teórica da linguagem PHP: conceitos, histórico, evolução e motivação para seu uso.
- Pesquisar e apresentar fundamentação teórica dos Padrões de Projeto: conceitos, classificação, tipos, motivação e exemplos.
- Desenvolver e apresentar uma aplicação modelo para demonstrar o uso de alguns padrões.

## 1.2 Estrutura do Trabalho

Este trabalho é composto por cinco capítulos. Neste primeiro capítulo é apresentado o tema e seus objetivos. No capítulo dois, são apresentados conceitos sobre a programação orientada a objetos. O capítulo três apresenta algumas linguagens de programação para o desenvolvimento WEB, dando ênfase à linguagem PHP que é uma das bases deste trabalho. No capítulo quatro estão

apresentados os padrões de projeto e os exemplos de suas aplicações com a linguagem PHP. O capítulo 5 é composto pelas conclusões, contribuições e sugestões de trabalhos futuros sobre o tema desta monografia.

# 2. DESENVOLVIMENTO ORIENTADO A OBJETOS

A programação orientada a objetos (POO) é um paradigma de desenvolvimento de software que se aproxima mais da realidade, onde a representação dos componentes do software é realizada através de objetos, como os objetos da realidade, que possuem características (atributos) e funções (métodos), pelas quais os objetos realizam ações ou mudam de estado. A POO possui algumas vantagens em relação à programação estruturada, como modularização, facilidade de extensão através de herança ou composição, maior facilidade de reutilização, entre outras. Para se entender a POO é necessário se ter em mente alguns conceitos, dentre os quais se podem destacar os seguintes:

## 2.1. Classes

Segundo Ricarte (2001), uma classe é um gabarito para a definição de objetos e descreve que propriedades, ou atributos, os objetos terão. As classes também definem quais as funções, ou métodos, os objetos serão capazes de executar. Ricarte (2001), também afirma que a definição de classes e seus inter-relacionamentos é o principal resultado da etapa de projeto de software. Para Zandstra (2006), uma classe é um modelo de código usado para gerar objetos. As definições de classes serão utilizadas durante o desenvolvimento para criar quantos objetos, ou instâncias de classe, forem necessários para atingir o objetivo do sistema. Como pode ser observado na figura 1, uma classe é representada na UML (Linguagem de Modelagem Unificada), como um retângulo dividido verticalmente em três partes: A identificação da classe, seus atributos e seus métodos.

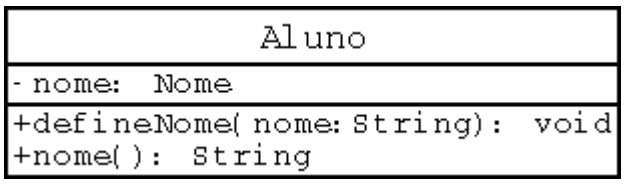


Fig. 1: Representação de uma classe na UML
Fonte: http://www.dca.fee.unicamp.br/cursos/PooJava/desenvolvimento/umlclass.html

## 2.2. Objetos

Objetos, segundo Ricarte (2001), são instâncias de classes e é através deles que praticamente todo o processamento ocorre em sistemas implementados com linguagens de programação orientadas a objetos. Zandstra (2006), afirma que um objeto são dados estruturados de acordo com o modelo definido em uma classe.

Um objeto é criado a partir da definição de uma classe, portanto possui todos os atributos e métodos definidos em sua classe base e das classes às quais sua classe base herdou através da herança e por isso é dito que o objeto é do tipo da classe. Durante a execução de uma aplicação, somente objetos são utilizados, sendo uma classe apenas um modelo para a construção destes objetos.

## 2.3. Herança

Herança, segundo LEITE E RAHAL JÚNIOR (2009), pode ser entendido como sendo um conjunto de instâncias criadas a partir de outro conjunto de instâncias com características semelhantes, e os elementos deste subconjunto herdam todas as características do conjunto original. Para Leite (2009), herança é o compartilhamento de atributos e operações entre classes com base em relações hierárquicas, ou seja, é a utilização de superclasses para criar as subclasses. O conceito de herança é muito importante para a reutilização de código, pois uma classe base pode ser utilizada para se construir várias outras mais específicas sem que seja necessário duplicar as características comuns. Na figura 2, pode-se observar a representação da herança na UML. A seta com a ponta “aberta” (sem preenchimento) sai da classe filha em direção à superclasse.

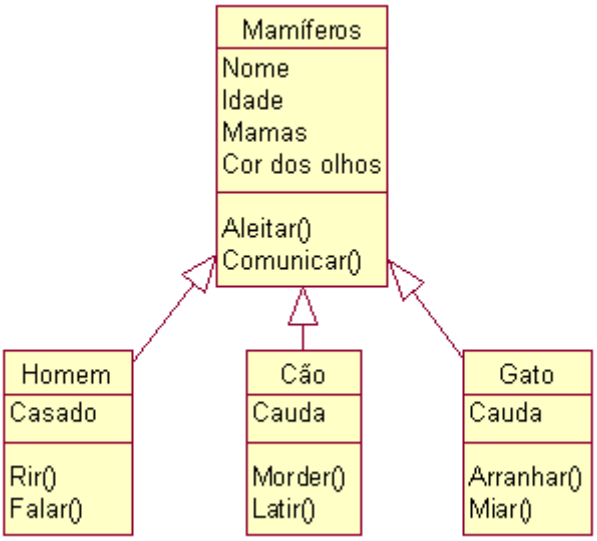


Figura 2: Esquema da Hierarquia de Classes

Fonte: http://www.ccuec.unicamp.br/revista/infotec/artigos/leite_rahal.html

### 2.4. Polimorfismo

Para Lemos (2009), polimorfismo é a capacidade que um mesmo método tem de poder se comportar de diferentes maneiras em diferentes classes. Segundo Ricarte (2001), polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. O conceito de polimorfismo é importante na programação orientada a objetos, pois permite que um método definido na superclasse possa ser redefinido na subclasse, se necessário, para atender às suas necessidades específicas. Para que o polimorfismo seja usado, a assinatura do método na subclasse deve ser exatamente igual à assinatura do método na superclasse, pois se a lista de argumentos for diferente, estará sendo usada a sobrecarga de métodos e não o polimorfismo.

### 2.5. Encapsulamento

Segundo LEITE E RAHAL JÚNIOR (2009), encapsulamento é a base de toda a abordagem da programação orientada a objetos. Isto porque contribui fundamentalmente para diminuir os malefícios causados pela interferência externa sobre os dados. A boa prática de programação orientada a objetos sugere que todos os atributos sejam definidos como privados (*private*), e que possam ser acessados apenas através de métodos públicos colocados dentro do objeto. Isto protege os atributos, encapsulando-os e também torna a interface do objeto mais clara. O encapsulamento permite criar classes totalmente independentes, fáceis de se manipular e de maior manutenibilidade, pois se a interface for mantida, todo o código interno à classe pode ser alterado sem que as classes clientes precisem tomar conhecimento.

# 3. LINGUAGENS DE PROGRAMAÇÃO PARA A WEB

Os sistemas de informação estão a cada dia sendo mais utilizados pela internet, funcionando com base na comunicação entre um cliente e um servidor WEB. Hoje existem inúmeras linguagens de programação para a WEB, cada uma com uma particularidade. Todas possuem vantagens e desvantagem e cada especialista defende sua linguagem preferida. Neste trabalho serão citadas apenas algumas linguagens e sem entrar na questão de qual é melhor ou pior.

## 3.1. ASP

Segundo a Wikipédia (2009), ASP (Active Server Page) é uma estrutura de programação (não uma linguagem, asp é um *framework*) em *script* que se utiliza de VBScript, JScript, PearScript ou Python processadas pelo lado servidor para geração de conteúdo dinâmico na WEB. Os arquivos ASP possuem a extensão .asp e são formados por código HTML com *scripts* ASP embutidos. Estes *scripts* são instruções que são executadas no servidor. Ao final da execução de um *script* pelo servidor o mesmo gera uma página HTML e a retorna à aplicação solicitante. Para que o servidor WEB seja capaz de executar os *scripts* ASP, ele precisa ter instalado um interpretador ASP.

Uma página ASP pode ser acessada de qualquer navegador, pois todos os *scripts* são executados no servidor e apenas código HTML é entregue ao navegador, o que também protege os *scripts*. Segundo Alecrim (2003), a linguagem ASP é parecida com a linguagem Visual Basic.

## 3.2. ASP .NET

Segundo SANTANA FILHO E ZARA (2002), ASP.Net é uma plataforma baseada na .Net *Framework* para o desenvolvimento de aplicações WEB que estão

armazenadas no *Internet Information Server* (IIS) e utiliza os protocolos da Internet, como *HyperText Transfer Protocol* (HTTP) e o *Simple Object Acces Protocol* (SOAP). Para a Microsoft (2009), ASP.Net é uma tecnologia gratuita que permite a criação de aplicativos WEB dinâmicos pelos programadores, podendo ser usado tanto para criar desde pequenos *sites* pessoais até grandes aplicativos WEB de nível empresarial.

SANTANA FILHO E ZARA (2002), também afirmam que o ASP.Net facilita o desenvolvimento e instalação de dois tipos de aplicações WEB. O primeiro tipo são aplicações baseadas em *WEB Forms* que incluem conteúdo dinâmico, incluindo páginas WEB que expõem uma interface para o usuário em um *browser*. O segundo tipo são *Web Services* que expõem sua funcionalidade para outras aplicações e para os clientes e permitem que eles troquem informações.

Macoratti (2003), afirma que alguns benefícios da ASP.Net são:

- Páginas ASP.Net são compiladas. Quando uma página ASP.Net é requisitada ela é compilada e vai para o cachê do servidor, sendo assim carregadas mais rapidamente.
- Páginas ASP.Net são construídas com controles de interface do lado do servidor.
- Por ser o ASP.Net parte integrante do .Net *Framework*, já possui disponível mais de 3000 classes que podem ser usadas nas aplicações. Existem classes para gerar imagens, enviar email, entre outras.
- O ASP.Net é totalmente orientado a objetos.
- Com o Visual Studio .Net, o ambiente integrado permite criar uma página apenas arrastando e soltando os controles no formulário WEB.

Para criar páginas em ASP.Net, pode-se desenvolver utilizando linguagens como Visual Basic, C#, ou C++. Na primeira vez que a página for executada ou alterada o código é compilado para um código intermediário, chamado MSIL (Microsoft *Intermediate Language*), que é entregue ao .Net *Framework*, o qual fará a conversão para a linguagem binária e em seguida executará o código. Macoratti (2003), também afirma que a conversão do código MSIL para a linguagem

binária é feita pelo CLR (*Common Language Runtime*), que gerencia todo o serviço necessário, como memória, tipo de dados, exceções, código, etc.

### 3.3. JSP

JSP é a abreviatura de *Java Server Page* e designa uma tecnologia desenvolvida para possibilitar a criação de páginas WEB com a linguagem de programação Java. Os arquivos em JSP possuem a extensão .jsp e possuem em sua estrutura *tags* HTML, entre as quais são inseridas as sentenças Java a serem executadas no servidor. A Construção de uma página JSP é bem semelhante à de uma página PHP ou ASP, como pode ser observado na figura 3, que apresenta uma página JSP bem simples e o esquema de conversão da página em um *Servlet*.

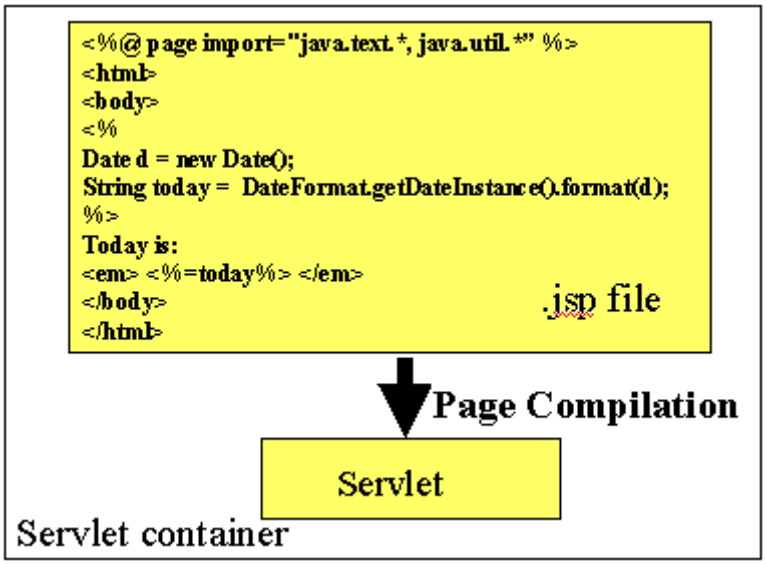


Figura 3: Exemplo de página JSP e do esquema de compilação de uma página JSP
Fonte: http://www.criarweb.com/artigos/227.php

Toda página JSP, durante seu primeiro acesso é transformada em um *Servlet*, o que faz com que esta etapa demore mais para ser respondida. A partir do segundo acesso a resposta é mais rápida, pois o servidor acessa diretamente o *Servlet* criado.

Para Lopes (2008), *Servlets* são classes Java para a WEB. Uma *Servlet* é uma classe Java que estende a classe javax.servlet.http.HttpServlet , porém não possui o método *main*. Isto ocorre porque as aplicações WEB realizam as solicitações não diretamente aos *Servlets*, como acontece normalmente com as classes Java, mas sim ao servidor WEB, que entrega a solicitação ao *Container* que a repassa ao *Servlet*, chamando seus métodos. Um dos *containers* mais utilizados é o TomCat, que é um software livre e de código aberto, escrito em Java. Ainda segundo Lopes (2008), a estrutura de um *servlet* possui o seguinte padrão:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
/* 99% de todos os servkets sai HttpServlet */
public class PadraoServlet extends HttpServlet{
   /* É aqui que seu Servlet consegue as referencias dos objetos solicitacao e resposta
    No mundo real 99% de todos os Servlet anulam ou o metodo doGet() ou o doPost()
    */
            public void doGet(HttpServletRequest request,
            HttpServletResponse response) throws IOException,
            ServletException{
         /* Utilize PrintWriter para escrever texto html no objeto de resposta*/
                        PrintWriter out = response.getWriter();
                      out.println(“Hello World”);
}}
```

Para ser acessível, uma *servlet* precisa estar configurada num arquivo XML chamado web.xml. Um arquivo web.xml pode declarar vários *servlets* e possui basicamente a seguinte estrutura:

```
<?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”>
            <!–chamando um Servlet  –>
            <servlet>
                        <servlet-name>servletteste</servlet-name>
                        <servlet-class>PadraoServlet</servlet-class>
            </servlet>

            <servlet-mapping>
                        <servlet-name>servletteste</servlet-name>
                        <url-pattern>/compras.do</url-pattern>
```

```
        </servlet-mapping>
</web-app>
```

### 3.4. PHP

Segundo o Manual do PHP (2003), PHP (*Hypertext Preprocessor*) é uma linguagem de programação de ampla utilização, interpretada, que é especialmente interessante para desenvolvimento para a Web e pode ser mesclada dentro do código HTML. Sua sintaxe lembra C, C++, Java e Perl.

Segundo Buyens (2000), o PHP é uma linguagem livre, licenciada conforme a Licença Pública GNU (GPL), ou seja, todo usuário possui a liberdade de executar, copiar, distribuir, estudar, modificar e aperfeiçoar seu software. Buyens (2002) também afirma que o código colocado em uma página Web para ser executado pelo PHP não faz parte da Licença Pública GNU. Uma grande vantagem do uso desta linguagem é a possibilidade de se encontrar inúmeros *scripts* prontos e testados na Internet, além de vários fóruns de discussões sobre o assunto. Outra vantagem é o fato do código ser executado no servidor, fazendo com que o cliente receba os resultados da execução dos *scripts*, sem nenhum modo de determinar como é o código fonte.

O Manual do PHP (2003), afirma que o PHP pode ser utilizado na maioria dos sistemas operacionais, incluindo Linux, várias variantes Unix (incluindo HP-UX, Solaris e OpenBSD), Microsoft Windows, Mac OS X, RISC OS e ainda é suportado pela maioria dos servidores Web atuais, incluindo Apache, Microsoft Internet Information Server, Personal Web Server, Netscape and iPlanet Servers, Oreilly Website Pro Server, Caudium, Xitami, OmniHTTPd, e muitos outros. Com o PHP, portanto, tem-se a liberdade para escolher o sistema operacional e o servidor Web, além de ser possível escolher entre utilizar programação estrutural ou programação orientada a objetos, ou ainda uma mistura deles.

Este trabalho apresentará a aplicação de alguns padrões de projeto utilizando a linguagem PHP, por isso serão descritas algumas características da linguagem.

### 3.4.1. A História do PHP

O PHP foi criado em 1994 por Rasmus Lerdof, um engenheiro de Software Canadiano-dinamarquês. A primeira versão do PHP foi desenvolvida para utilização pessoal, auxiliando Lerdof a monitorar as pessoas que acessavam o seu site pessoal. Em 1995, Lerdof montou um pacote chamado *Personal Home Page Tools* em resposta à crescente demanda de usuários do PHP. A versão 2 foi lançada sob o título de PHP/FI, incluindo o *Form Interpreter*, que era uma ferramenta para analisar sintaticamente consultas de SQL.

Em 1997, o PHP já era utilizado em aproximadamente 50000 sites em todo o mundo e já contava com uma pequena equipe de desenvolvimento que mantinha o projeto com contribuições de desenvolvedores e usuários em todo o mundo.

A partir da versão 3 do PHP, dois programadores israelenses, chamados Zeev Suuraski e Andi Gutmans reescreveram completamente a arquitetura da linguagem, porém os objetos ainda não eram considerados uma parte necessária da nova sintaxe, embora tenha sido adicionado, já nesta versão, um pequeno suporte a classes.

Em 1998 já haviam mais de 100000 domínios utilizando o PHP, número que ultrapassaria um milhão de domínios um ano mais tarde.

Para a versão 4 do PHP, o mecanismo Zend (nome derivado de Zeeve e Andi), que movia a linguagem, foi escrito do zero, segundo Zandstra (2006), melhorando o suporte a objetos, porém ainda deixando algumas falhas, como a atribuição de objetos à variáveis, que ainda se dava por valores e não por referência. Assim cada atribuição de um objeto à uma variável, a passagem do mesmo a uma função ou o retorno desta causava a cópia do objeto.

Segundo Zandstra (2006), o PHP 5 representa, explicitamente, o endosso aos objetos e à programação orientada a objetos. O PHP continua não sendo uma linguagem orientada a objetos, porém já permite a utilização do pleno potencial da orientação a objetos. Este trabalho utilizará o suporte a objetos do PHP 5 para aplicar os padrões de projeto.

### 3.4.2. A Orientação a Objetos e o PHP

#### *3.4.2.1. Classes no PHP*

Na linguagem PHP, a definição de uma classe é realizada conforme o exemplo abaixo. O nome da classe é precedido da palavra chave *class* e os atributos e os métodos podem vir precedidos das palavras chave *private*, *protected* ou *public*, que definem o escopo, ou visibilidade destes. Segundo Zandstra (2006), propriedades e métodos públicos podem ser acessados a partir de qualquer contexto. Propriedades e métodos privados só podem ser acessados de dentro da classe (nem mesmo subclasses têm acesso) e propriedades e métodos *protected* só podem ser acessados de dentro da classe ou de uma subclasse. Ainda segundo Zandstra (2006), o nome da classe pode ter qualquer combinação de números e de letras, embora não deva começar com um número.

```
Class NomedaClasse {
    Private $Atributo_1;
    Protected $Atributo_2;
    Public $Atributo_3;

    Public function __construct($Argumento_1, $Argumento_2) {
            $this->Atributo_1 = $Argumento_1;
            $this->Atributo_2 = $Argumento_2;
    }

    Public function NomeFuncao ($Argumento_1, $Argumento_2)  {
            Corpo da Função
            Return $ValorRetorno;
    }
}
```

#### *3.4.2.2. Objetos no PHP*

Em PHP, um objeto é instanciado através do operador *new* seguido do nome da classe e de parênteses, como $Objeto = new NomeClasse();. Para acessar atributos de um objeto, utiliza-se os caracteres -> precedidos do nome do objeto ($Objeto) e sucedidos do nome do atributo sem o caracter especial $, ficando a chamada ao atributo assim: $Objeto->Atributo. Os métodos são acessados de forma semelhante, da seguinte forma: $Objeto->Método_1().

#### *3.4.2.3. Herança no PHP*

Em PHP, a herança é utilizada colocando-se a palavra chave *extends* após o nome da classe filha na sua definição seguida pelo nome da superclasse, da seguinte forma:

```
Class NomeClasseFilha extends NomeSuperClasse {
    Corpo da Classe
}
```

Em PHP não existe herança múltipla, ou seja, uma classe somente pode estender uma única classe base. Para acessar atributos ou métodos da classe base, ou superclasse, é possível fazê-lo utilizando a palavra chave *parent* seguida por "::" e pelo nome do atributo ou método desejado. Um axemplo seria parent::método_exemplo();

#### *3.4.2.4. Classes Abstratas no PHP*

Uma classe abstrata é sempre iniciada pela palavra chave *abstract*, como pode ser observado no exemplo abaixo, e não pode ser instanciada, ou seja, não é possível criar objetos a partir da definição de uma classe abstrata. Se uma classe possui pelo menos um método abstrato também deve ser definida como abstrata.

```
abstract class ClasseAbstrata {
        // Força a classe que extende (a subclasse) a definir esse método
        abstract protected function pegarValor();
        abstract protected function valorComPrefixo( $prefixo );

        // Método comum
        public function imprimir() {
                echo “Valor:”. $this->pegarValor();
        }
}
```

Métodos definidos como abstratos não podem ter implementação. Estes devem apenas definir a assinatura do método, como é apresentado no exemplo acima. Uma classe herdeira de uma classe abstrata deve implementar todos os métodos definidos como abstratos na classe base abstrata.

### *3.4.2.5. Interfaces no PHP*

Segundo o manual do PHP (2009), interfaces de objetos permitem a criação de código que especifica quais métodos e variáveis uma classe deve implementar, sem ter que definir como estes métodos serão tratados. Interfaces devem ser definidas usando a palavra chave “*Interface*” e todos os seus métodos devem ser declarados como *public*, além de não poderem ter seu conteúdo definido. No exemplo abaixo pode ser observada a estrutura de uma interface.

```
interface MinhaInterface {
        public function setVariavel($nome, $valor);
        public function Mostrar($Variavel);
}
```

Para implementar uma interface deve ser usado o operador *implements* e todos os métodos da interface devem ser implementados. No exemplo abaixo pode ser observada uma implementação.

```
// Implementa a interface
class MinhaClasse implements MinhaInterface {
            $variaveis;
             public function setVariavel($nome, $valor) {
                        $this-> variaveis [$nome] = $ valor;
             }
public function Mostrar ($Variavel) {
             foreach($this-> variaveis as $ nome => $ valor) {
                          echo “Nome: ”. $nome.”<br>”;
                          echo “Valor: ”. $valor.”<br>”;
            }
   }
}
```

# 4. PADRÕES DE PROJETO

Para Zandstra (2006), um padrão de projeto é um problema analisado e uma boa prática para sua solução explicada. Deschamps (2009), define um padrão de projeto como uma estrutura recorrente no projeto de software orientado a objetos. Problemas que se repetem são comuns, e para Zandstra (2006), padrões de projeto descrevem e formalizam esses problemas e suas soluções, tornando a experiência, obtida a duras penas, disponível para uma comunidade maior de programação.

O termo padrões de projeto foi utilizado pela primeira vez pelo arquiteto Christopher Alexander, na década de 70, para definir padrões relacionados à sua área de atividade. Somente em 1987, segundo a Wikipédia (2009), os programadores Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto para a área da ciência da Computação, apresentando alguns padrões para a construção de janelas na linguagem Smalltalk. Ainda segundo a Wikipédia (2009), o movimento ao redor de padrões de projeto ganhou popularidade com o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, publicado em 1995. Os autores desse livro são Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos como a “Gangue dos Quatro”.

Padrões de projeto não podem ser vistos como receitas, onde é possível segui-las passo a passo para atingir um objetivo. Um padrão de projeto deve ser visto como uma experiência documentada de forma genérica, na qual o contexto deve ser analisado e o padrão adaptado a este.

Para Zandstra (2006), um padrão de projeto consiste em quatro partes:

- Nome: O nome é importante para facilitar a identificação do padrão e deve balancear brevidade e descrição. Fowler (2003) afirma que nomes de padrões são cruciais, porque parte do propósito dos padrões é criar um vocabulário que permita ao desenvolvedor se comunicar mais eficazmente.
- O problema: Para Zandstra (2006), reconhecer um problema é mais difícil do que aplicar as soluções de um catálogo de padrões. A dificuldade de identificar um problema e sua solução em um catálogo de padrões justifica o cuidado que se deve ter ao se descrever e se contextualizar o problema.

- A solução: A solução é resumida em conjunto com o problema e detalhada, muitas vezes usando diagramas de interação e de classes UML.
- Conseqüências: Tudo possui conseqüências, e com os padrões de projeto não é diferente, assim há sempre considerações a serem feitas antes de se tomar uma decisão. Para Deschamps (2009), esta etapa analisa os resultados, vantagens e desvantagens obtidos com a aplicação do padrão.

## 4.1. Categorias dos Padrões de Projeto

Em geral, os padrões de projeto são classificados em três tipos: padrões de criação, padrões estruturais e padrões comportamentais.

### 4.1.1. Padrões de Criação

Os padrões de criação se preocupam com a instanciação de objetos. Segundo Nelson e Nelson (2009), padrões de criação abstraem o processo de instanciação, ajudam a tornar o sistema independente de como objetos são criados, encapsulam conhecimento sobre quais classes concretas o sistema usa e esconde como instâncias dessas classes são criadas.

A GoF (*Gang of Four*), apresenta cinco padrões de criação, sendo eles: *Singleton*, *Abstract Factory*, *Builder*, *Factory Method* e *Prototype*.

#### *4.1.1.1. O Padrão Singleton*

Segundo Zandstra (2006), o padrão *Singleton* é uma classe especial que gera uma e somente uma instância de objeto. Para Ricarte (2006), o objetivo do

*Singleton* é assegurar que uma classe tem uma única instância e estabelecer um ponto de acesso global para essa instância. Ricarte (2006), ainda afirma que este padrão é adequado para lidar com recursos que devem ser únicos em um sistema, como um controlador de um dispositivo ou um gerenciador. Como o padrão *Singleton* garante que só exista uma instância da classe, temos a certeza que todos os objetos que utilizam uma instância desta classe usam a mesma instância.

A estrutura do padrão *Singleton* é bastante simples, composto por somente uma classe, como pode ser observado na figura abaixo:

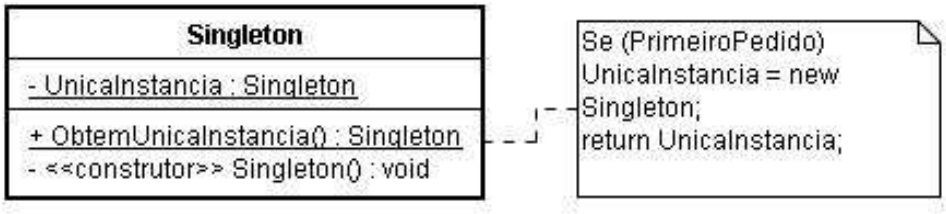


Figura 4: Estrutura do Padrão Singleton
Fonte: Padrões de Projeto. Alcides Calsavara

Para exemplificar o padrão *singleton*, será utilizada uma classe chamada ConexaoMySQL, pela qual sua única instância será responsável por realizar todas as conexões necessárias com o banco de dados na execução de um *script*. Abaixo podem ser observadas as classes AbstractConexao, que fornece a interface para a criação de conexões com bancos de dados e a classe ConexaoMySQL, que fornece a implementação para conexões com bancos de dados MySQL. Neste caso está sendo aplicado o princípio “Programe para uma interface e não para uma implementação”, pois este princípio permite que seja fácil, caso necessário, utilizar uma classe ConexaoSQLServer que possua a mesma interface para realizar conexão com um banco de dados SQL Server no futuro.

```
abstract class Conexao {

        private $ArrayObservadores;
        public $Servidor;
        public $Usuario;
        private $Senha;
        private $BancoDados;
        public $Sql;
        private $Link;
```

```
	private $Resultado;

	public static function getInstance() {
		if(empty(self::$Instance)) {
		self::$Instance = new ConexaoMySQL();
		}
		return self::$Instance;
	}
	abstract function setPropriedade($Chave, $Valor);
	abstract function getPropriedade($Chave);
	abstract function Conecta();
	abstract function executaQuery($query);
	abstract function Desconecta();
	abstract function IncluirObservador($Observador);
	abstract function RemoverObservador($Observador);
	abstract function NotificarObservadores();
}

class ConexaoMySQL implements InterfaceObservado {

	public Static $Instance;

	private function __construct() {
	}

	public static function getInstance() {
		if(empty(self::$Instance)) {
		self::$Instance = new ConexaoMySQL();
		}
		return self::$Instance;
	}

	public function setPropriedade($Chave, $Valor) {
		$this->$Chave = $Valor;
	}
	public function getPropriedade($Chave) {
		return $this-> $Chave;
	}
	public function Conecta() {
		$this->Link=mysql_connect($this->Servidor,$this->Usuario,$this->Senha);
		if(!$this->Link) {
				echo "Falha na conexao com o Banco de Dados!<br />";
				echo "Erro: " . mysql_error();
				die();
		}
```

```
                elseif(!mysql_select_db($this->BancoDados, $this->Link)) {
                        echo "O Banco de Dados solicitado não pode ser aberto!<br />";
                        echo "Erro: " . mysql_error();
                        die();
                }
        }
        //Esta função executa uma Query
        public function executaQuery($query) {
                $this->Conecta();
                $this->Sql=$query;
                if($this->Resultado=mysql_query($this->Sql)) {
                $this->Desconecta();
                $this->NotificarObservadores();
                return $this->Resultado;
                }
                else {
                        echo "Ocorreu um erro na execução da SQL";
                        echo "Erro :" . mysql_error();
                        echo "SQL: " . $query;
                        die();
                        $this->Desconecta();
                }
        }
        //Esta função desconecta do Banco
        public function Desconecta() {
                return mysql_close($this->Link);
        }

        public function IncluirObservador($Observador) {
                $this->ArrayObservadores[strlen($this->ArrayObservadores)+1] = $Observador;
        }

        public function RemoverObservador($Observador) {

        }

        public function NotificarObservadores() {
                foreach($this->ArrayObservadores as $Observador) {
                        $Observador->Update($this);
                }
        }
}
```

Como o construtor da classe ConexaoMySQL é privado, não podemos chamá-lo para instanciar um objeto desta classe, por isso devemos utilizar o método getInstance(), que é público e estático. Para chamarmos o método estático getInstance utilizamos a seguinte notação: ConexaoMySQL :: getInstance(). Abaixo um exemplo da instanciação do Singleton ConexaoMySQL.

```
<?php
include("ConexaoMySQL.class.php");
$Conexao = ConexaoMySQL :: getInstance();
$Conexao->setPropriedade("servidor","127.0.0.1");
$Conexao->setPropriedade("Usuario","meuusuario");
$Conexao->setPropriedade("Senha","minhasenha");
$Conexao->setPropriedade("BancoDados","meubancodedados ");
}
?>
```

### *4.1.1.2. O Padrão Factory Method*

Para Deschamps (2009), o padrão *Factory Method* define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe será instanciada, permitindo a uma classe postergar a instanciação de subclasses. Zandstra (2006), afirma que o padrão *Factory Method* aborda o problema de como criar instâncias de objetos quando seu código enfoca tipos abstratos. Como a programação orientada a objetos enfatiza a utilização de classes abstratas, *Factory Method* ajuda a resolver este problema. Abaixo é possível observar a estrutura do padrão.

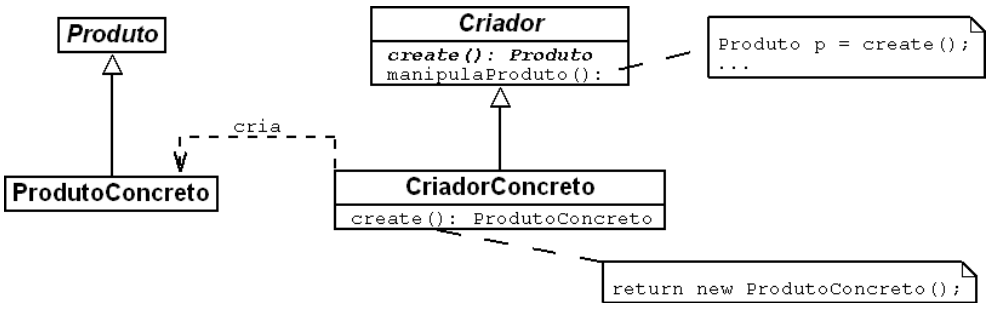


Figura 5 : Estrutura do Padrão *Factory Method*

Fonte: Padrões de Software. Ivan L. M. Ricarte

Na figura 6, pode ser observado o diagrama de classes da aplicação exemplo utilizando o *Factory Method*. Na figura podemos observar a classe abstrata FabricaEvento (criador), que é responsável por definir a utilização do *Factory Method*, que neste exemplo será o método CriaEvento(). Os criadores concretos, que são FabricaEventosMG e FabricaEventosES, são responsáveis por implementar o método CriaEvento() de acordo com as particularidades de cada região (MG ou ES).

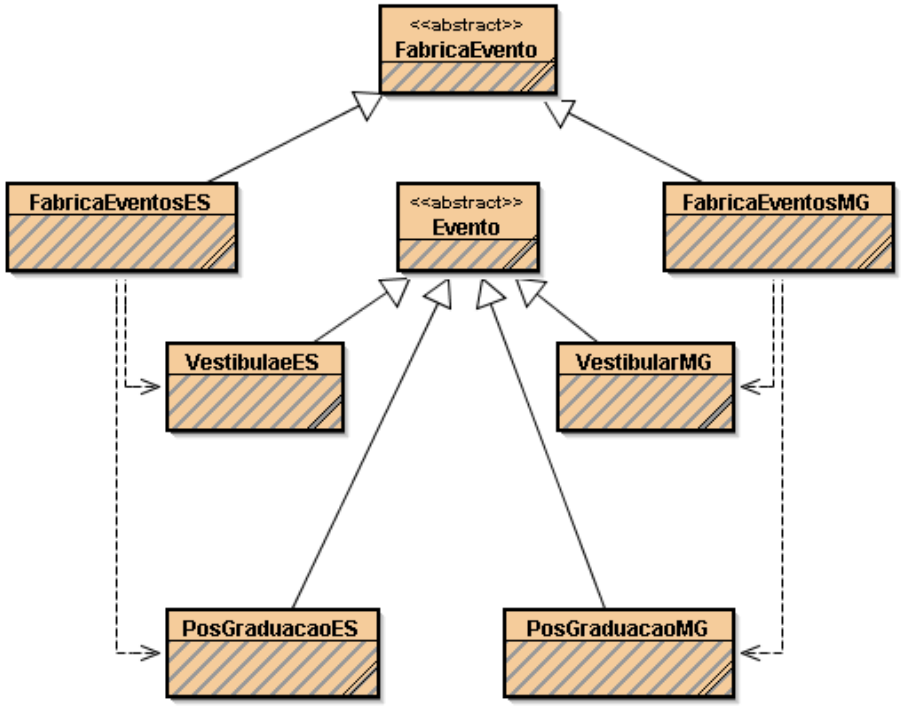


Figura 6: Diagrama de classes da estrutura *Factory Method* da aplicação exemplo

Abaixo podem ser observadas as implementações de algumas das classes exemplo:

```
// Classe abstrata que define a utilização do Factory Method (Criador Abstrato)
abstract class FabricaEvento {
        abstract function CriaEvento($Tipo);
}

//  Classe concreta que implementa o Factory Method CriaEvento (Criador Concreto)
require_once("AbstractFabricaEvento.class.php");
require_once("ItensEvento.class.php");
require_once("Vestibular.class.php");
require_once("PosGraduacao.class.php");
```

```
class FabricaEventosMG extends FabricaEvento{

	private $Evento;

	public function __construct(){
		echo "Criou uma fábrica de eventos MG<br>";
	}

	public function CriaEvento($Tipo){
		if($Tipo == "Vestibular"){
		$this->Evento = new Vestibular(new ItensEvento());
		return $this->Evento;
		}
		if($Tipo == "PosGraduacao"){
		$this->Evento = new PosGraduacao(new ItensEvento());
		return $this->Evento;
		}
	}
}

// Classe abstrata base para a criação dos eventos
abstract class Evento {

	private $Codigo;
	private $Nome;
	private $Periodo;
	private $Ano;
	private $Unidade;
	private $Campus;
	public $FabricaItens;

	public $ArrayCursos;
	public $ArrayCandidatos;

	public function setPropriedade($Chave, $Valor) {
		$this->$Chave = $Valor;
	}
	public function getPropriedade($Chave) {
		return $this-> $Chave;
	}

	abstract function Preparar();

	public function CriarIteradorCursos() {
		return new Iterador($this->ArrayCursos);
```

```
	}

	public function CriarIteradorCandidatos() {
		return new Iterador($this->ArrayCandidatos);
	}

	final function FazerClassificacao() {

		$this->CarregarCursos();
		$this->CarregarCandidatos();
		$this->ClassificarCandidatos();
	}

	public function CarregarCursos() {
		echo "Carregando Cursos ...<br>";
	}

	public function CarregarCandidatos() {
		echo "Carregando Candidatos ...<br>";
	}

	abstract function ClassificarCandidatos();

}
// Classe Concreta que implementa um Vestibular com as particularidades de MG
class VestibularMG extends Evento{

	private $ArraySalas;
	private $ArrayAplicadores;
	private $Prova;

	public function __construct($ArgumentoFabricaItens){
			echo "Criou Um Vestibular<br>";
		$this->FabricaItens = $ArgumentoFabricaItens;
	}

	public function Preparar(){

		$this->ArrayCursos = $this->FabricaItens->CriaCurso("CursoGraduacao",$QdeCursos);
		$this->ArrayCandidatos = $this->FabricaItens->CriaCandidato($QdeCandidatos);
		$this->ArraySalas = $this->FabricaItens->CriaSala($QdeSalas);
		$this->ArrayAplicadores = $this->FabricaItens->CriaAplicadorProva($QdeAplicadores);
		$this->ArrayProva = $this->FabricaItens->CriaProva($QdeProvas);
	}
```

```
        public function Corrigir() {
                echo "Corrigindo ...";
        }

        public function ClassificarCandidatos() {
                echo "Classificando candidatos de acordo com as regras do vestibular ...<br>";
        }

        public function CriarIteradorSalas() {
                return new Iterador($this->ArraySalas);
        }

        public function CriarIteradorAplicadores() {
                return new Iterador($this->ArrayAplicadores);
        }

}
```

Como pode ser observado, pode-se criar novos eventos apenas implementando a classe abstrata evento e incluindo sua instanciação no método fábrica CriaEvento().

#### *4.1.1.3. O Padrão Abstract Factory*

Para FREEMAN E FREEMAN (2007), o padrão *Abstract Factory* fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Zandstra (2006), afirma que o padrão *Abstract Factory* aborda o problema de criar fábricas que produzam conjuntos relacionados de classes em grandes aplicações.

Em padrões de projeto, muitas vezes um padrão trabalha associado a outro, e é o que acontece com os padrões *Factory Method* e *Abstract Factory*, onde geralmente cada método no *Abstract Factory* é implementado como um *Factory Method*.

Na figura 7 pode ser observada a estrutura do padrão *abstract Factory*.

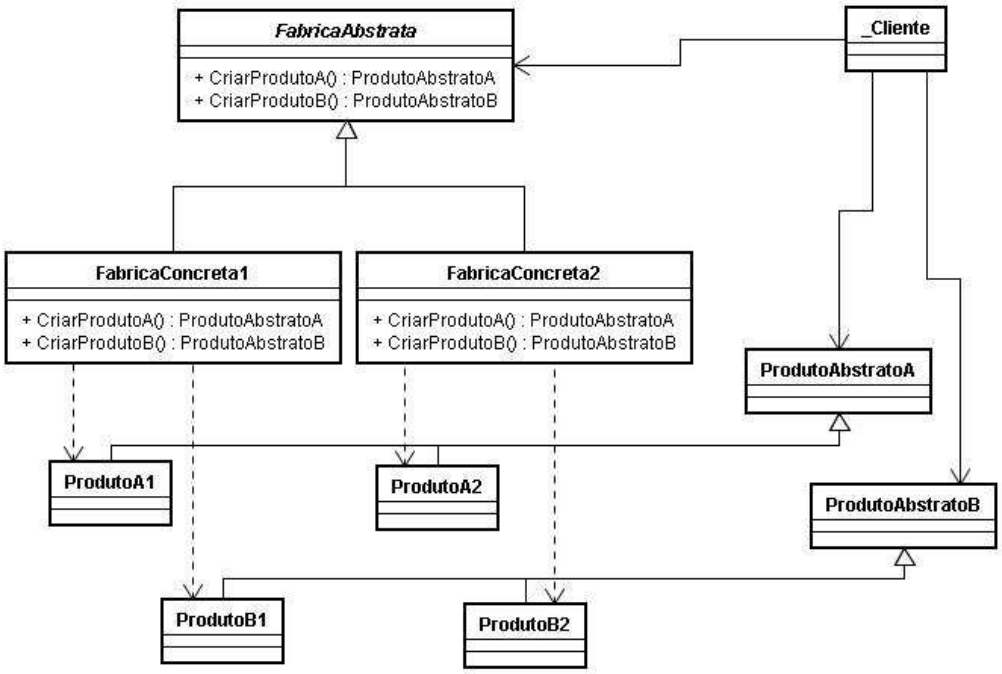


Figura 7: Estrutura do padrão *Abstract Factory*

Fonte: Padrões de Projeto. http://www.ppgia.pucpr.br/~alcides

Na figura 8 pode ser observado o diagrama de classes da implementação do padrão *Abstract Factory* na aplicação exemplo.

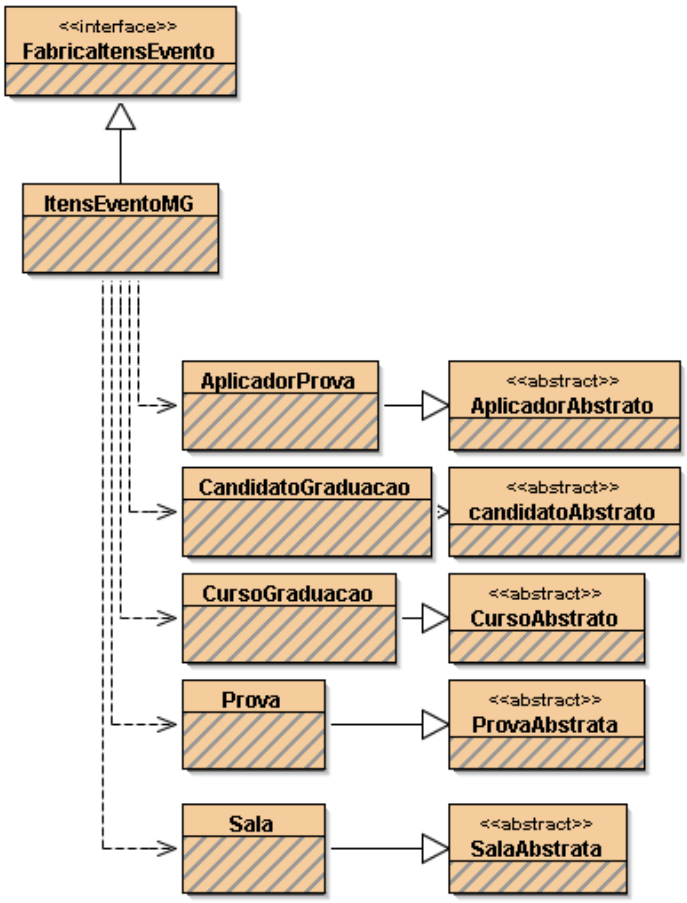


Figura 8: Diagrama de Classes que implementa o padrão *Abstract Factory*

A interface FabricaItensEvento fornece a interface para a criação dos itens que formam os eventos da aplicação exemplo. Nela estão definidas as interfaces dos métodos CriaCurso(), CriaCandidato(), CriaAplicadorProva(), CriaProva() e CriaSala(). A implementação em PHP desta interface pode ser observada abaixo:

```
interface FabricaItensEvento {

        public function CriaCurso($Tipo,$Qde);
        public function CriaSala($Qde);
        public function CriaAplicadorProva($Qde);
        public function CriaProva($Qde);
        public function CriaCandidato($Qde);
}
```

A classe ItensEvento implementa a interface. O método CriaCurso(), é implementado como um *Method Factory* e pode criar cursos de Graduação ou Pós

Graduação, decidido em tempo de execução, de acordo com o contexto. Abaixo pode ser observada a implementação da classe ItensEvento em PHP.

```
class ItensEvento implements FabricaItensEvento{

	private $Item;

	public function __construct(){
		echo "Criou uma fábrica de itens de eventos<br>";
	}

	public function CriaCurso($Tipo,$Qde){

		if($Tipo == "CursoGraduacao"){
			for($i=0; $i < $Qde; $i++){
				$this->Item[$i] = new CursoGraduacao();
			}
			return $this->Item;
		}

		if($Tipo == "CursoPosGraduacao"){
			for($i=0; $i < $Qde; $i++){
				$this->Item[$i] = new CursoPosGraduacao();
			}
			return $this->Item;
		}
	}

	public function CriaSala($Qde){
			for($i=0; $i < $Qde; $i++){
				$this->Item[$i] = new Sala();
			}
			return $this->Item;
	}

	public function CriaAplicadorProva($Qde){
			for($i=0; $i < $Qde; $i++){
				$this->Item[$i] = new AplicadorProva();
			}
			return $this->Item;
	}

	public function CriaProva($Qde){
			for($i=0; $i < $Qde; $i++){
```

```
                        $this->Item[$i] = new Prova();
                }
                return $this->Item;
        }

        public function CriaCandidato($Qde){
                for($i=0; $i < $Qde; $i++){
                        $this->Item[$i] = new Candidato();
                }
                return $this->Item;
        }

}
```

### 4.1.2. Padrões Estruturais

Para Deschamps (2009), padrões estruturais tratam da forma como classes e objetos estão organizados para a formação de estruturas maiores. Ricarte (2006), afirma que padrões estruturais tem o objetivo de isolar do cliente como objetos estão associados.

Segundo a *Gang of Four*, os padrões estruturais podem ser divididos em sete, sendo eles: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Façade*, *Flyweight* e *Proxy*. Neste trabalho serão apresentados apenas alguns destes padrões.

#### *4.1.2.1. O Padrão Composite*

Segundo Zandstra (2006), o padrão *Composite* é uma forma de agregar e, depois, gerenciar grupos de objetos semelhantes, de modo que um objeto individual não pode ser distinto para um cliente a partir de uma coleção de objetos. Para Calsavara (2009), o padrão *Composite* permite construir objetos complexos através de uma composição recursiva que define uma árvore de objetos, onde todos os objetos são acessados de maneira consistente e homogênea, já que todos possuem uma superclasse ou uma interface comum. Freemam e Freemam (2007), afirmam

que com o padrão *Composite* os clientes podem tratar objetos individuais ou composições de objetos de maneira uniforme.

Na figura 9 pode ser observada a estrutura do padrão *Composite*.

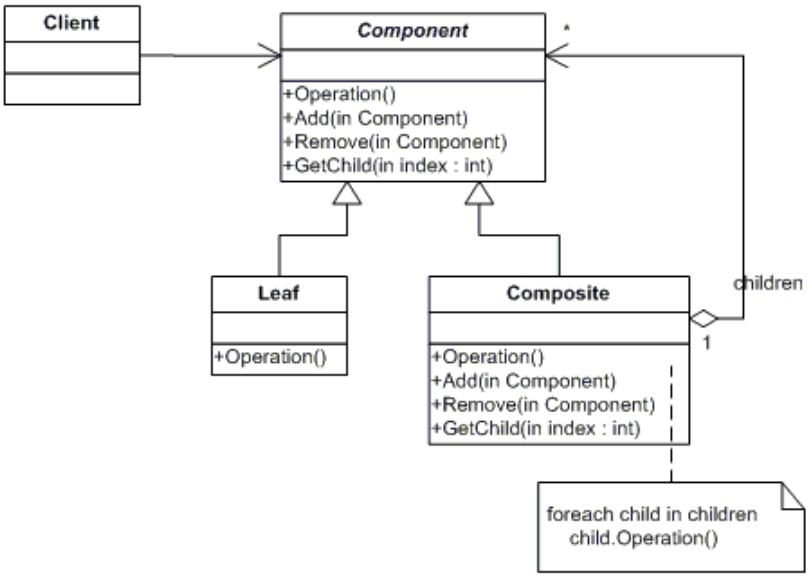


Figura 9: Estrutura do Padrão *Composite*

Fonte: Padrões de Projetos. Deschamps

O padrão *Composite* permite que sejam criados objetos compostos por objetos do mesmo tipo dispostos em uma árvore e esta estrutura é bastante útil. O menu da aplicação exemplo foi construído utilizando o padrão *Composite* e o diagrama UML do exemplo pode ser observado na figura 10.

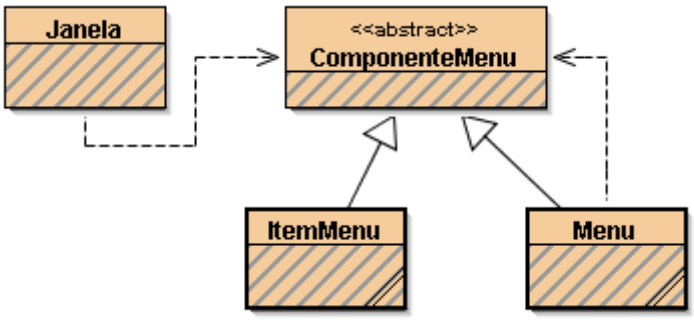


Figura 10: Diagrama UML do Menu da Aplicação Exemplo

Abaixo podem ser observadas as implementações das principais classes apresentadas no diagrama da figura 10.

```
abstract class ComponenteMenuAbstrato {
	public function Adiciona($ComponenteMenu) {
		throw new Exception('Não é possível adicionar Componentes de Menu a folhas');
	}
	public function Remove($ComponenteMenu) {
		throw new Exception('Não é possível remover Componentes de Menu de folhas');
	}
	public function RecuperaComponenteMenu($i) {
		throw new Exception('Não é possível recuperar Componentes de Menu de folhas');
	}
	public function RecuperarNome() {
		throw new Exception('Não é possível recuperar o nome do Menu');
	}
	public function RecuperaDescricao() {
		throw new Exception('Não é possível recuperar a descrição do Menu');
	}
	public function RecuperaLinkMenu($ComponenteMenu) {
		throw new Exception('Não é possível remover Componentes de Menu de folhas');
	}
	public function Mostrar() {
		throw new Exception('Não é possível mostrar o Menu');
	}
}

class ItemMenu extends ComponenteMenuAbstrato {

	public $Nome;
	public $Descricao;
	public $Link;

	public function __construct($Nome,$Descricao,$Link) {
		$this->Nome = $Nome;
		$this->Descricao = $Descricao;
		$this->Link = $Link;
	}
	public function RecuperarNome() {
		return $this->Nome;
	}
	public function RecuperaDescricao() {
		return $this->Descricao;
	}
	public function RecuperaLinkMenu($ComponenteMenu) {
		return $this->Link;
	}
	public function Mostrar() {
```

```
                    echo "<font color='#000000' face='Verdana, Arial, Helvetica, sans-serif' size='2px'>    <a name='".$this->Nome."' href='".$this->Link."' >".$this->Nome."</a></font><br>";
                }
            }

            class Menu extends ComponenteMenuAbstrato {

                public $ArrayComponentes;
                public $Nome;
                public $Descricao;

                public function __construct($Nome,$Descricao) {
                    $this->Nome = $Nome;
                    $this->Descricao = $Descricao;
                }
                public function Adiciona($ComponenteMenu) {
                    $this->ArrayComponentes[$ComponenteMenu->Nome] = $ComponenteMenu;
                }
                public function Remove($ComponenteMenu) {
                    foreach($this->ArrayComponentes as $Chave => $Valor){
                        if($Chave == $ComponenteMenu->Nome){
                            $this->ArrayComponentes[$ComponenteMenu->Nome] = NULL;
                        }
                    }
                }
                public function RecuperaComponenteMenu($Nome) {
                    foreach($this->ArrayComponentes as $Chave => $Valor){
                        if($Chave == $Nome){
                            return $this->ArrayComponentes[$Chave];
                        }
                    }
                }
                public function RecuperarNome() {
                    return $this->Nome;
                }
                public function RecuperaDescricao() {
                    return $this->Descricao;
                }
                public function Mostrar() {
                    echo "<strong><font color='#000000' face='Verdana, Arial, Helvetica, sans-serif' size='2px'>".$this->Nome."</font></strong><br>";
                    foreach($this->ArrayComponentes as $Chave){
                        echo "    ";
                        $Chave->Mostrar();
```

```
            }
        }
    }
```

Para se exemplificar a utilização do menu construído a partir da estrutura do padrão *composite*, pode-se observar no script PHP abaixo a criação de um menu para a aplicação exemplo.

```
<?php
require_once("AbstractComponenteMenu.class.php");
require_once("Menu.class.php");
require_once("ItemMenu.class.php");

$Menu1 = new Menu("Vestibular","Este é o Menu Referente aos Vestibulares");
$ItemMenu1 = new ItemMenu("Criar Vestibular","Direciona à página de Cadastro de Vestibulares","pagina.php?Pagina=centro.php");
$ItemMenu2 = new ItemMenu("Alterar Vestibular","Direciona à página de Alteração de Vestibulares","cadastros/AlterarVestibular.php");
$ItemMenu3 = new ItemMenu("Excluir Vestibular","Direciona à página de Exclusão de Vestibulares","cadastros/ExcluirVestibular.php");
$Menu1->adiciona($ItemMenu1);
$Menu1->adiciona($ItemMenu2);
$Menu1->adiciona($ItemMenu3);
$Menu2 = new Menu("Pós Graduação","Este é o Menu Referente às Pós Graduações");
$ItemMenu4 = new ItemMenu("Criar Pós Graduação","Direciona à página de Cadastro de Pós Graduações","cadastros/CadastrarPosGraduacao.php");
$ItemMenu5 = new ItemMenu("Alterar Pós Graduação","Direciona à página de Alteração de Pós Graduações","cadastros/AlterarPosGraduacao.php");
$ItemMenu6 = new ItemMenu("Excluir Pós Graduação","Direciona à página de Exclusão de Pós Graduações","cadastros/ExcluirPosGraduacao.php");
$Menu2->adiciona($ItemMenu4);
$Menu2->adiciona($ItemMenu5);
$Menu2->adiciona($ItemMenu6);

$MenuGeral = new Menu("Eventos","Este é o Menu Geral da Aplicação");
$MenuGeral->adiciona($Menu1);
$MenuGeral->adiciona($Menu2);

$MenuGeral->Mostrar()

?>
```

*4.1.2.2. O Padrão Decorator*

O padrão *Decorator* possui uma estrutura semelhante à do padrão *Composite*, porém ajuda a modificar a funcionalidade de componentes concretos. Segundo Zandstra (2006), ao invés de usar apenas herança para resolver o problema da variação de funcionalidade, o padrão *Decorator* usa composição e delegação. Freemam e Freemam (2007), afirmam que o padrão *Decorator* anexa responsabilidades adicionais a um objeto dinamicamente, fornecendo uma alternativa flexível de subclasse para estender a funcionalidade.

Como pode ser observado na figura 11, um *decorator* possui a interface do componente ao qual irá estender suas funcionalidades, pois é criado através de herança, e também é composto por uma instância deste componente.

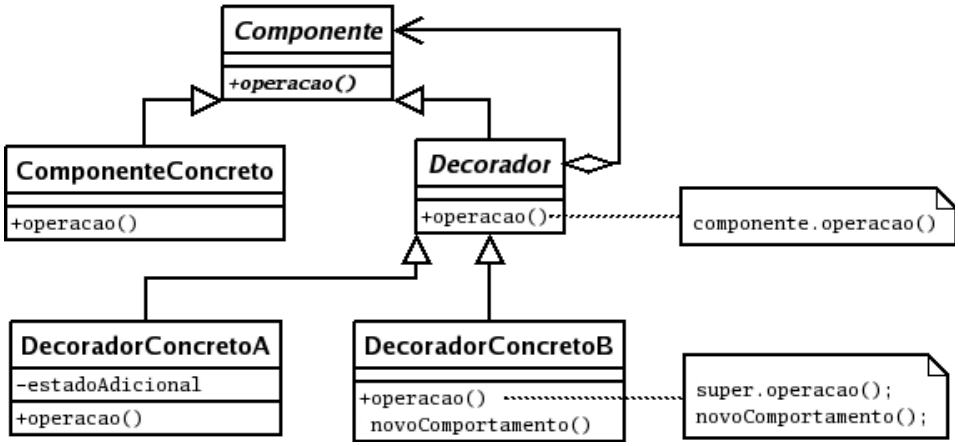


Figura 11: A estrutura do padrão *Decorator*

Fonte: Padrões de Projeto. Ricarte.

É possível utilizar um ou mais decoradores para cada objeto. Neste caso, o primeiro decorador engloba o objeto a ser decorado, o segundo decorador engloba o primeiro decorador e assim sucessivamente. Como os objetos são decorados dinamicamente em tempo de execução, podem-se utilizar quantos decoradores forem necessários.

Na figura 12 pode ser observado o diagrama de classes da construção das interfaces da aplicação exemplo, que utiliza o padrão *Decorator* para adicionar funcionalidades às páginas de interface.

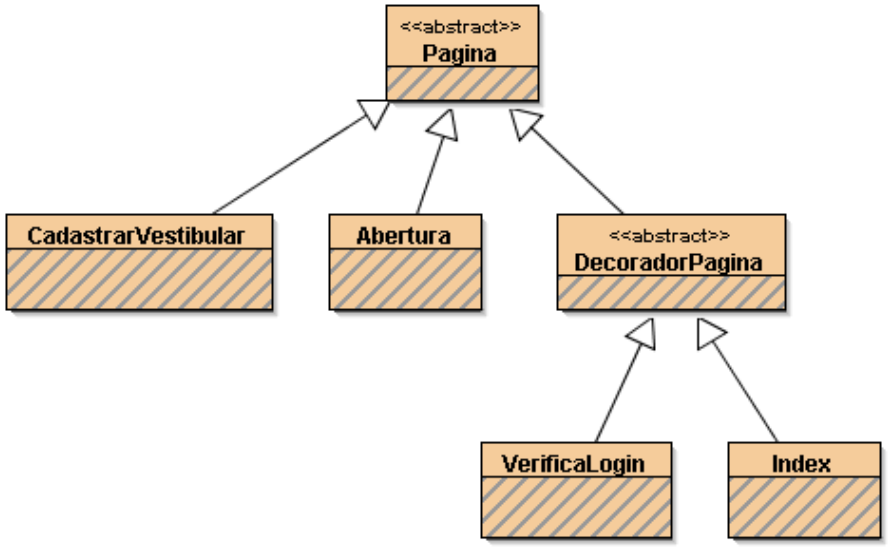


Figura 12: Diagrama de classes das interfaces da aplicação exemplo

A classe abstrata Pagina é a base para a construção das interfaces do sistema. Esta classe possui o método abstrato Corpo, que deve ser implementado em todas as interfaces que a estenderem. A implementação da classe abstrata Pagina pode ser observada abaixo:

```
abstract class Pagina {
    abstract function Corpo();
}
```

A classe Abertura, estende a classe abstrata Pagina e apenas exibe uma mensagem de boas vindas ao sistema, como pode ser observada em sua implementação:

```
class Abertura extends Pagina {

    public function Corpo() {
            echo "<html>";
            echo "<Body>";
            echo "<table>";
            echo "    <tr height='40'>";
            echo "            <td></td>";
            echo "    </tr>";
            echo "    <tr>";
            echo "            <td>Bem vindo ao sistema de gerenciamento de eventos<br>
```

```
                                                    Aqui você poderá realizar o gerenciamento de
todos os eventos cadastrados.<br><br>
                                                    Selecione uma das opções do Menu ao lado.
                                                </td>";
                        echo "    </tr>";
                        echo "</table>";
                        echo "</body>";
                        echo "</html>";
                }

        }
```

A classe Abstrata DecoradorPagina, é a interface para a utilização de decoradores para incrementar as páginas da aplicação. Esta classe é composta de um objeto do tipo página, que será o objeto a ser decorado. Cada implementação da classe DecoradorPagina, implementará a funcionalidade a ser acrescentada à página simples. A implementação da classe DecoradorPagina pode ser observada abaixo:

```
abstract class DecoradorPagina extends Pagina {

    protected $PaginaaDecorar;
    public function __construct($Pagina) {
            $this->PaginaaDecorar = $Pagina;
    }
}
```

A classe Index estende a classe abstrata DecoradorPagina, acrescentando às páginas simples que lhe forem passadas pelo construtor Um cabeçalho, um menu (já visto neste trabalho) na sua lateral esquerda e um rodapé na parte inferior da página. Abaixo pode ser observada a implementação da classe Index:

```
class Index extends  DecoradorPagina {

    public $Titulo;
    public $Topo; //<img src='Vestibular/imagens/Topo.GIF' width='100%' height='100%' />
    public $Menu;
    public $Centro;
    public $Rodape; //<img src='Vestibular/imagens/Rodape.GIF' />
```

```
        public function __construct($Titulo,$Topo,$Menu,$PaginaCentro,$Rodape) {
                $this->Titulo = $Titulo;
                $this->Topo = $Topo;
                $this->Menu = $Menu;
                $this->Centro = $PaginaCentro;
                $this->Rodape = $Rodape;

        }

        public function Corpo() {

                echo "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN' 'http://www.w3.org/TR/html4/loose.dtd'>";
                echo "<html>";
                echo "<head>";
                echo "<title>".$this->Titulo."</title>";
                echo "<meta http-equiv='Content-Type' content='text/html; charset=iso-8859-1'>";
                echo "</head>";
                echo "<body>";
                echo "    <table width='800' height='600' border='1'>";
                echo "          <tr height='80'>";
                echo "                <td width='100%' align='left' valign='middle' colspan='2'>".$this->Topo."</td>";
                echo "          </tr>";
                echo "          <tr height='500' align='left' valign='top'>";
                echo "                <td width='200' height='500'>";
                                        $this->Menu->Mostrar();
                echo "                </td>";
                echo "                <td width='600' height='500' valign='top'>";
                                        $this->Centro->Corpo();
                echo "                </td>";
                echo "          </tr>";
                echo "          <tr height='20'>";
                echo "                <td colspan='2' align='center'>".$this->Rodape."</td>";
                echo "          </tr>";
                echo "    </table>";
                echo "</body>";
                echo "</html>";
        }
}
```

A classe VerificaLogin, que também estende a classe DecoradorPagina, é outro exemplo de decorador. Esta classe também recebe um objeto do tipo pagina,

que pode ser uma página simples como um objeto da classe abertura ou uma página já decorada, como um objeto da classe Índex. A classe VerificaLogin possui a capacidade de decorar o objeto recebido com uma verificação se existe ou não uma sessão de usuário no sistema. Caso exista uma sessão de usuário, é chamado o método Corpo() do objeto recebido, caso contrário o Browser redireciona o usuário para uma página existente no atributo Redireciona do objeto VerificaLogin. Abaixo pode ser observada a implementação da classe VerificaLogin:

```
ob_start();
session_start();

require_once("AbstractDecoradorPagina.class.php");

class VerificaLogin extends  DecoradorPagina {

    public $Pagina;
    public $Sessao;
    public $Redireciona;

    public function __construct($Pagina) {
            $this->Pagina = $Pagina;
    }
    public function SetVerificador($Sessao,$LinkRedireciona) {
            $this->Sessao = $Sessao;
            $this->Redireciona = $LinkRedireciona;
    }

    private function Verifica(){
            // Verifica se existe os dados da sessão de login
            if(!(isset($_SESSION[$this->Sessao])))
            {
                    // Usuário não logado! Redireciona para a página de login
                    header("Location: ".$this->Redireciona);
                    exit;
            }
    }

    public function Corpo() {
            $this->Verifica();
            $this->Pagina->Corpo();
    }
}
```

#### *4.1.2.3. O Padrão Façade*

O padrão *Façade* é utilizado para simplificar a interface de subsistemas mais complexos e é muito útil para facilitar o entendimento e utilização destes subsistemas pelos clientes. A utilização de um subsistema é encapsulada dentro de uma interface, ou fachada, que acessa as classes utilizadas pela aplicação e chama os métodos necessários, não tendo o cliente que acessar diretamente estes.

FREEMAN E FREEMAN (2007), afirmam que o padrão *Façade* fornece uma interface unificada para um conjunto de interfaces de um subsistema, definindo uma interface de nível mais alto que facilita a utilização do subsistema. Na figura 13 pode-se observar a estrutura do padrão *Façade*.

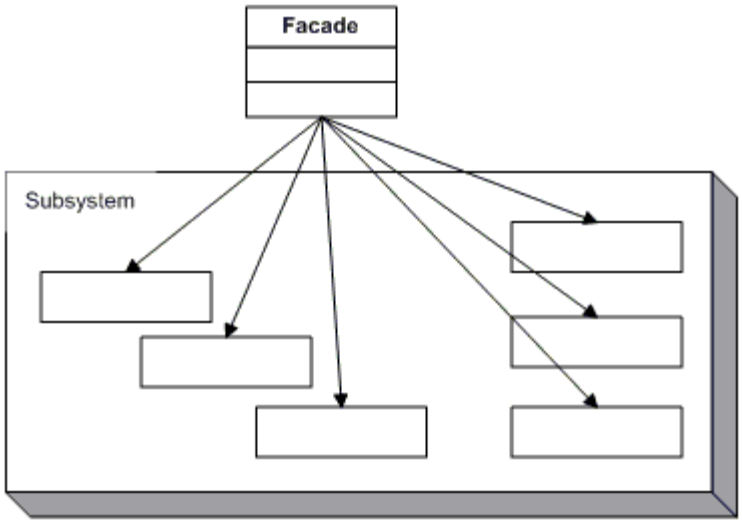


Figura 13: Estrutura do padrão *Facade*
Fonte: Padrões de Projeto – Uma Introdução. Deschamps

Na aplicação exemplo o padrão *Façade* foi utilizado para facilitar a configuração do sistema e a montagem do menu, que é utilizado por toda a aplicação. Em apenas uma classe, a Fachada, o cliente pode configurar o sistema sem precisar acessar métodos de várias classes, ficando todo este trabalho concentrado na interface da Fachada. Abaixo pode ser observado o diagrama de classes da utilização do padrão Façade na aplicação exemplo.

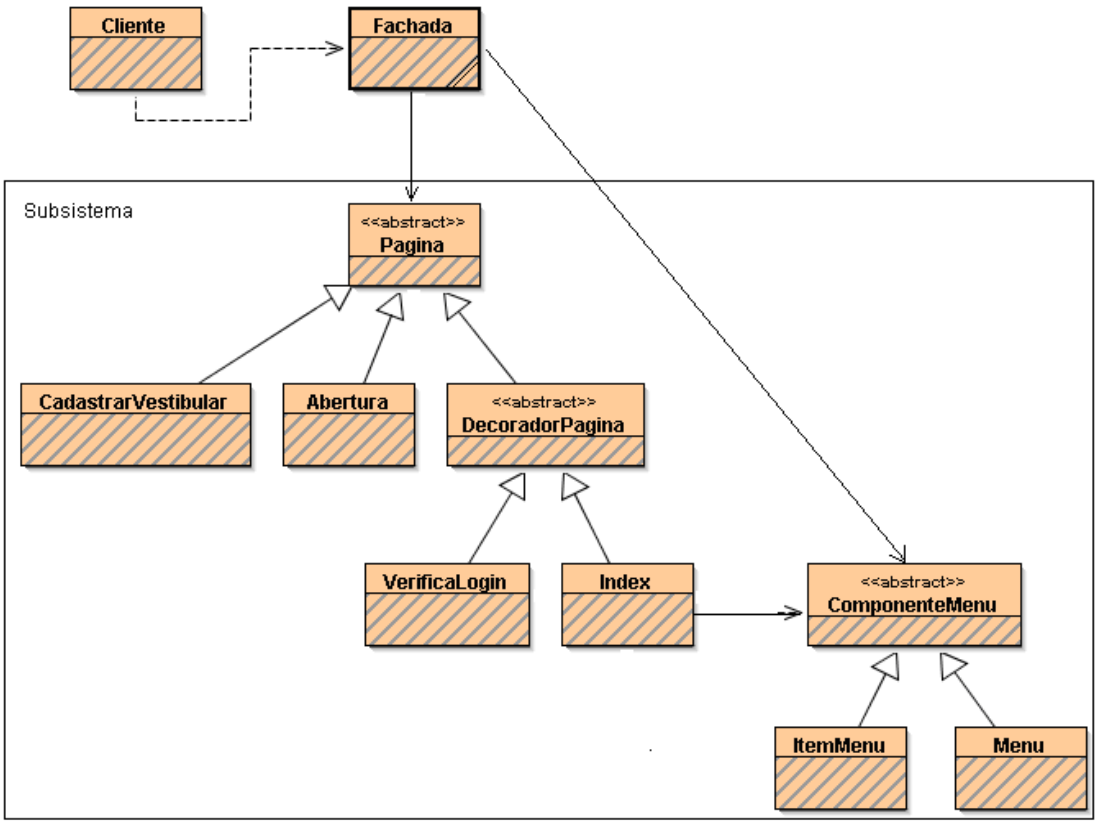


Figura 14: Aplicação do padrão Facade na aplicação exemplo

Abaixo pode-se observar a implementação da classe Fachada. Ela reúne em uma única interface as principais funções do subsistema. Caso se necessite de acessar algum método que não foi simplificado pela fachada, não há problemas, pois as classes do subsistema continuam existindo.

```
class Fachada {

    private $Titulo;
    private $Topo;
    private $MenuGeral;
    private $PaginaCentro;
    private $Rodape;
    private $Sessao;
    private $LinkRedireciona;

    public function __construct($Titulo,$Topo,$PaginaCentral,$Rodape,$Sessao,$LinkRedireciona,$MenuGeral) {
        $this->Titulo = $Titulo;
        $this->Topo = $Topo;
```

```
                $this->PaginaCentro = new $PaginaCentral();
                $this->MenuGeral = $MenuGeral;
                $this->Rodape = $Rodape;
                $this->Sessao = $Sessao;
                $this->LinkRedireciona = $LinkRedireciona;
        }

        public function MontaMenu($Menu,$Nome,$Descricao,$Link) {

                if($Menu == "" & $Link == "")
                        $this->MenuGeral->adiciona(new Menu($Nome,$Descricao));
                else if($Menu != "" & $Link == "")
                        $this->MenuGeral->adiciona(new Menu($Nome,$Descricao));
                else
                        $this->MenuGeral->ArrayComponentes[$Menu]->adiciona(new
ItemMenu($Nome,$Descricao,$Link));
        }

        public function MontaPagina() {

                $Pagina = new Index($this->Titulo,$this->Topo,$this->MenuGeral,$this-
>PaginaCentro,$this->Rodape);
                $Login = new VerificaLogin($Pagina);
                $Login->SetVerificador($this->Sessao,$this->LinkRedireciona);
                $Login->Corpo();

        }
}
```

Para facilitar a visualização da simplificação que o padrão *Façade* pode trazer, o código PHP abaixo configura, e monta a estrutura básica da aplicação exemplo.

```
<?php

    require_once("Vestibular/Fachada.class.php");
    require_once("Vestibular/Paginas/CriarVestibular.class.php");
    require_once("Vestibular/Paginas/AlterarVestibular.class.php");
    require_once("Vestibular/Paginas/ExcluirVestibular.class.php");
    require_once("Vestibular/Paginas/CriarPosGraduacao.class.php");
    require_once("Vestibular/Paginas/AlterarPosGraduacao.class.php");
    require_once("Vestibular/Paginas/ExcluirPosGraduacao.class.php");
```

```
$Titulo = "Metodista de Minas";
$Topo = "<img src='Vestibular/imagens/Topo.GIF' width='100%' height='100%' />";
if(isset($_GET["Pagina"]))
	$PaginaCentral = $_GET["Pagina"];
else
	$PaginaCentral = "Abertura";
$MenuGeral;
$Rodape = "<img src='Vestibular/imagens/Rodape.GIF' />";
$Sessao = "Email";
$LinkRedireciona = "http://www.metodistademinas.edu.br";

$Menu1 = new Menu("Eventos","Este é o Menu Geral da Aplicação");

$teste = new Fachada($Titulo,$Topo,$PaginaCentral,$Rodape,$Sessao,$LinkRedireciona,$Menu1);
$teste->MontaMenu("Eventos","Vestibular","Este é o Menu Referente aos Vestibulares",""); // Nome, Descricao, Link
$teste->MontaMenu("Vestibular","Criar Vestibular","Direciona à página de Cadastro de Vestibulares","testefachada.php?Pagina=CriarVestibular"); // Nome, Descricao, Link
$teste->MontaMenu("Vestibular","Alterar Vestibular","Direciona à página de Alteração de Vestibulares","testefachada.php?Pagina=AlterarVestibular"); // Nome, Descricao, Link
$teste->MontaMenu("Vestibular","Excluir Vestibular","Direciona à página de Exclusão de Vestibulares","testefachada.php?Pagina=ExcluirVestibular"); // Nome, Descricao, Link
$teste->MontaMenu("Eventos","PosGraduacao","Menu Principal",""); // Nome, Descricao, Link
$teste->MontaMenu("PosGraduacao","Criar Pos Graduacao","Direciona à página de Cadastro de PosGraduacao","testefachada.php?Pagina=CriarPosGraduacao"); // Nome, Descricao, Link
$teste->MontaMenu("PosGraduacao","Alterar Pos Graduacao","Direciona à página de Alteração de PosGraduacao","testefachada.php?Pagina=AlterarPosGraduacao"); // Nome, Descricao, Link
$teste->MontaMenu("PosGraduacao","Excluir Pos Graduacao","Direciona à página de Exclusão de PosGraduacao","testefachada.php?Pagina=ExcluirPosGraduacao"); // Nome, Descricao, Link
$teste->MontaPagina();

?>
```

A figura 15 apresenta a interface gerada com o código acima. Como a chamada ao código foi realizada sem a passagem do parâmetro Pagina, a página central, ou seja, a página apresentada à direita do Menu foi a página de Abertura, com a mensagem inicial. Caso se selecione um dos itens do menu, será mudada apenas esta página central, que pode ser, neste caso: CriarVestibular, AlterarVestibular, ExcluirVestibular, CriarPosGraduacao, alterarPosGraduacao ou ExcluirPosGraduacao. Para se incluir itens no Menu basta acrescentar novos itens através do método MontaMenu() da classe fachada, passando como parâmetros o

nome do menu pai, ou a qual o menu será um submenu, O nome que será apresentado, a descrição do menu e um link caso o menu seja um item de menu clicável.

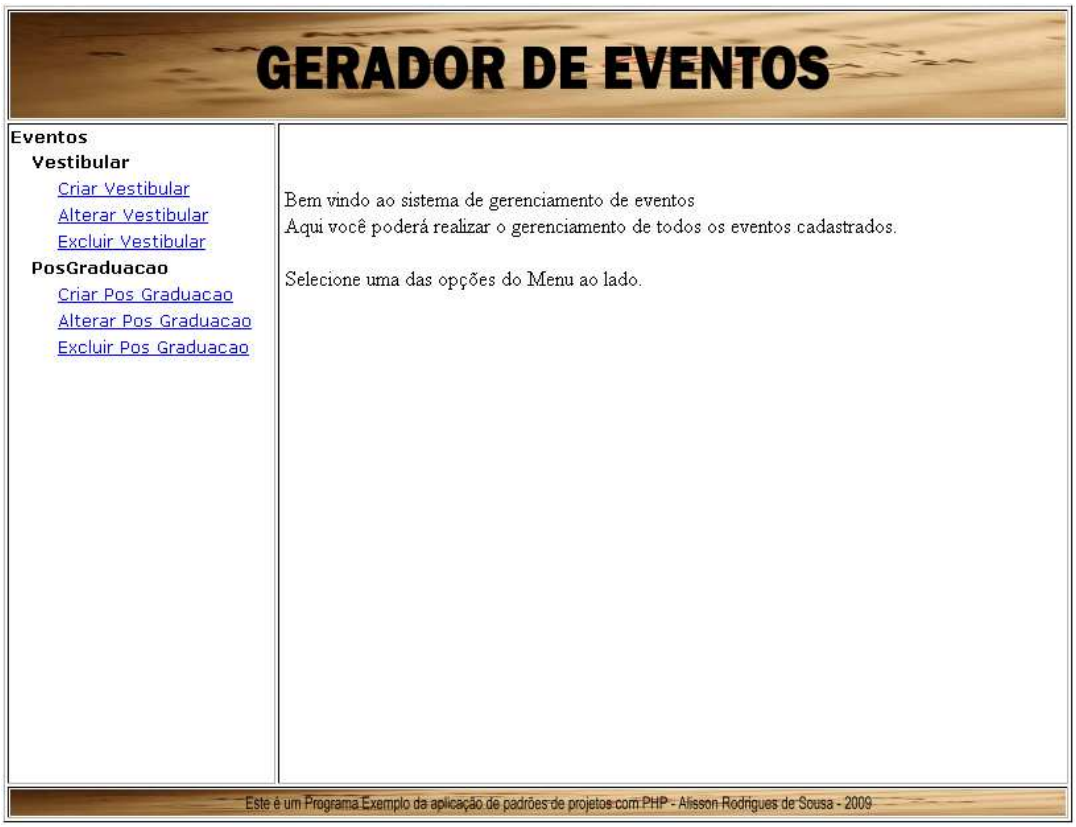


Figura 15: Interface gerada com a utilização do padrão *Facade*

### 4.1.3. Padrões Comportamentais

Para Ricarte (2006), o objetivo dos padrões comportamentais é isolar do cliente a atribuição de responsabilidades e formas de execução das operações. Os padrões comportamentais cuidam da forma como os algoritmos atribuem as responsabilidades entre os objetos. Fernandes (2003), afirma que enquanto os padrões comportamentais se concentram apenas na forma como os objetos são conectados, criam complexos fluxos de controle que são difíceis de seguir em *runtime*. Para a *Gang of Four*, existem onze padrões comportamentais, sendo eles; *Chain of Responsability*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento,*

*Observer*, *State*, *Strategy*, *Template Method* e *Visitor*. Este trabalho irá apresentar alguns destes padrões.

#### *4.1.3.1. O Padrão Iterator*

Segundo FREEMAN E FREEMAN (2007), o padrão *Iterator* fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor a sua representação subjacente. Ricarte (2006), afirma que o padrão *Iterator* isola como os objetos agregados são sequencialmente percorridos. O padrão *Iterator* permite que elementos agregados em um objeto sejam acessados sem que seja necessário expor a representação interna destes elementos. Uma particularidade do PHP, é que por o PHP não ser uma linguagem tipada, uma única classe *Iterador* pode ser usada para qualquer tipo de elemento agregado. Na figura 16 pode ser observada a estrutura do padrão *Iterator*.

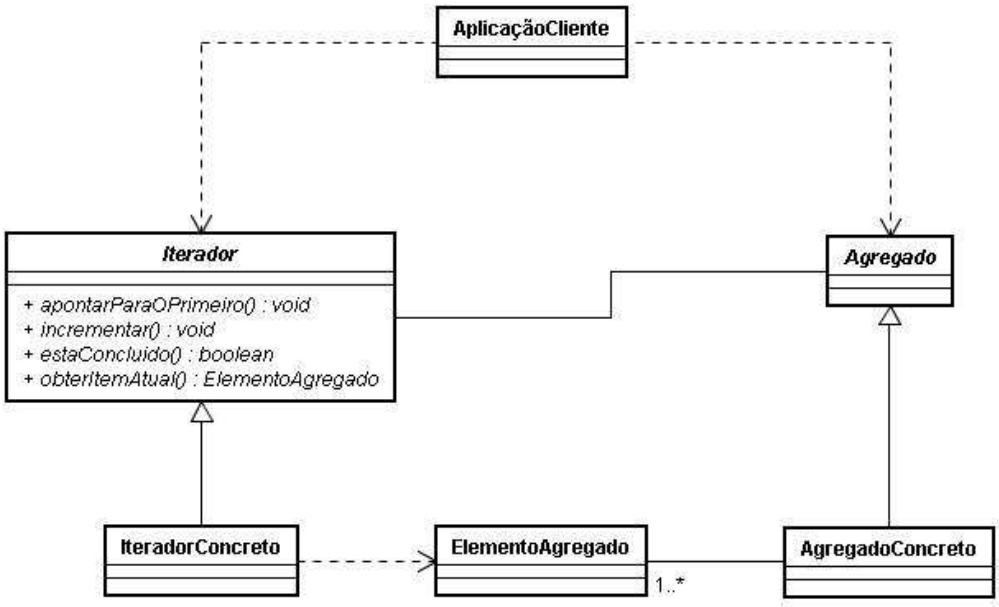


Figura 16: Estrutura do Padrão Iterator.
Fonte: Padrões de Projeto. Alcides Calsavara

Na aplicação exemplo, objetos de uma única classe *iterador* são usados para acessar os objetos Candidatos, Salas e Aplicadores de Provas que estão

agregados na classe Vestibular. Abaixo pode ser observado o diagrama UML da aplicação do padrão *Iterator* na aplicação exemplo.

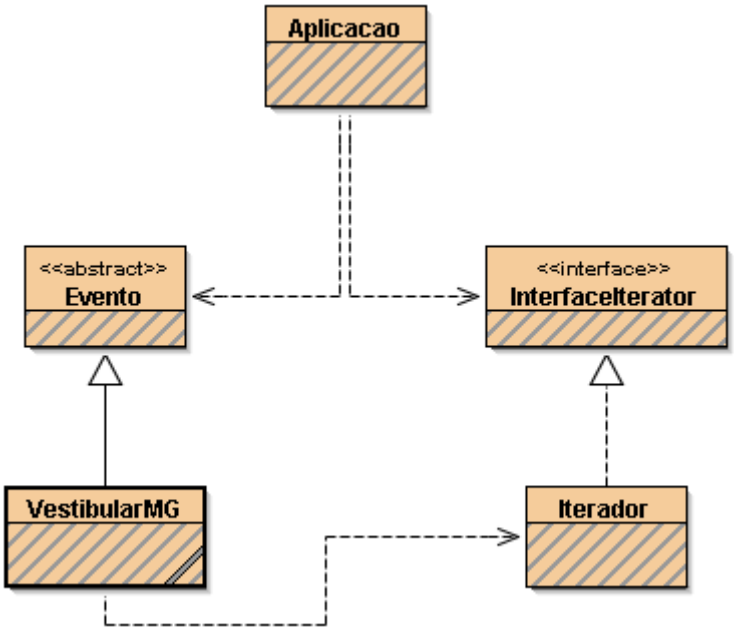


Figura 17: Diagrama de Classes da Aplicação do Padrão *Iterator*.

Abaixo pode ser abservada a implementação da classe *Iterador*.

```
interface InterfaceIterador {
    public function ExisteProximo();
    public function Proximo();
}

class Iterador implements InterfaceIterador {

    public $Array;
    public $Posicao;
    public $Objeto;

    public function __construct($Array) {
            echo "Criou Iterador<br>";
            $this->$Array = $Array;
            $this->Posicao = 0;
    }

    public function Proximo(){
            $this->Objeto = $this->Array[$this->Posicao];
            $this->Posicao = $this->Posicao + 1;
```

```
            return $this->Objeto;
        }

        public function ExisteProximo() {
            if($this->Posicao > strlen($this->Array) || $this->Array[$this->Posicao] == "") {
                return false;
            }
            else {
                return true;
            }
        }
    }
```

O código PHP abaixo exemplifica a utilização do padrão Iterator na aplicação exmplo:

```
<?php

require_once("Vestibular/ItensEvento.class.php");
require_once("Vestibular/VestibularMG.class.php");
require_once("Vestibular/Iterador.class.php");

$Vestibular = new VestibularMG(new ItensEvento());
$Vestibular->Preparar();

$IteradorCandidatos = $Vestibular->CriarIteradorCandidatos();
echo "<br>Candidatos Inscritos No Vestibular MG<br><br>";
while($IteradorCandidatos->ExisteProximo()){
    $Candidato = $IteradorCandidatos->Proximo();
    $Candidato->Mostra();
}

$IteradorSalas = $Vestibular->CriarIteradorSalas();
echo "<br>Salas cadastradas no Vestibular MG<br><br>";
while($IteradorSalas->ExisteProximo()){
    $Sala = $IteradorSalas->Proximo();
    $Sala->Mostra();
}

$IteradorAplicadores = $Vestibular->CriarIteradorAplicadores();
echo "<br>Aplicadores de Provas cadastrados no Vestibular MG<br><br>";
while($IteradorAplicadores->ExisteProximo()){
    $Aplicador = $IteradorAplicadores->Proximo();
    $Aplicador->Mostra();
```

```
}

?>
```

*4.1.3.2. O Padrão Observer*

Segundo Ricarte (2006), um observador define uma interface comum para que um sujeito possa notificar todos os observadores interessados que estejam cadastrados com ele. FREEMAN E FREEMAN (2007), afirmam que o padrão *Observer* define a dependência um-para-muitos entre objetos para que quando um objeto mude de estado todos os seus dependentes sejam avisados e atualizados automaticamente. FREEMAN E FREEMAN (2007), afirmam ainda que quando dois objetos estão levemente ligados, podem interagir, mas sabem muito pouco um do outro. Neste contexto o padrão *observer* fornece um *design* de objeto onde os sujeitos e os observadores são levemente ligados. A figura 18 apresenta a estrutura do padrão *Oserver*.

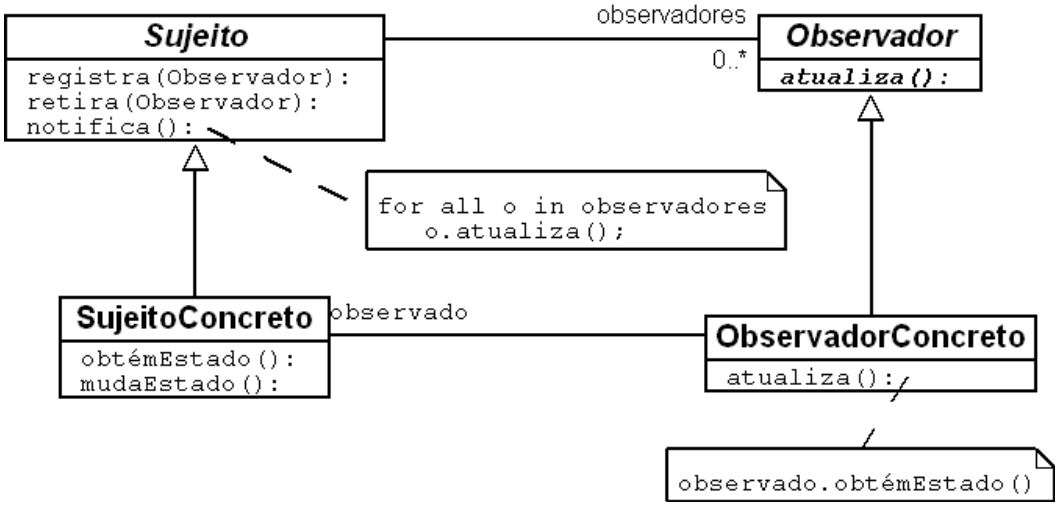


Figura 18: Estrutura do Padrão *Observer*

Fonte: Padrões de Projeto. Ivan Ricarte

Na aplicação exemplo, o padrão *Observer* foi utilizado para registrar um *log* das execuções de *querys* no banco de dados. Na figura 19 pode ser observado o diagrama de classes do *Observer* implementado na aplicação exemplo.

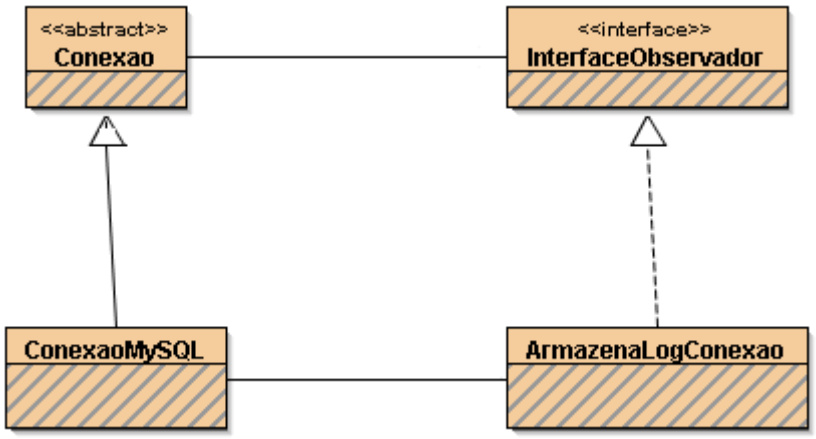


Figura 19: Estrutura do Padrão Observer na aplicação exemplo

As implementações das classes utilizadas na aplicação exemplo que aplicam o padrão *Observer* podem ser observadas abaixo:

```
interface InterfaceObservador {
    function update($Observador);
}

class ArmazenaLogConexao implements InterfaceObservador {
    public function Update($Observado) {
        $Data = date("d/m/y");
        $Hora = date("h:m:s");
        echo "Usuario ".$Observado->Usuario." logou dia ".$Data." às ".$Hora." e executou a query '".$Observado->Sql."'<br>";
    }
}

interface InterfaceObservado {
    function IncluirObservador($Observador);
    function RemoverObservador($Observador);
    function NotificarObservadores();
}

class ConexaoMySQL implements InterfaceObservado {

    public Static $Instance;

    private function __construct() {
    }
```

```
public static function getInstance() {
        if(empty(self::$Instance)) {
        self::$Instance = new ConexaoMySQL();
        }
        return self::$Instance;
}

public function setPropriedade($Chave, $Valor) {
        $this->$Chave = $Valor;
}
public function getPropriedade($Chave) {
        return $this-> $Chave;
}
public function Conecta() {
        $this->Link=mysql_connect($this->Servidor,$this->Usuario,$this->Senha);
        if(!$this->Link) {
                echo "Falha na conexao com o Banco de Dados!<br />";
                echo "Erro: " . mysql_error();
                die();
        }
        elseif(!mysql_select_db($this->BancoDados, $this->Link)) {
                echo "O Banco de Dados solicitado não pode ser aberto!<br />";
                echo "Erro: " . mysql_error();
                die();
        }
}
//Esta função executa uma Query
public function executaQuery($query) {
        $this->Conecta();
        $this->Sql=$query;
        if($this->Resultado=mysql_query($this->Sql)) {
        $this->Desconecta();
        $this->NotificarObservadores();
        return $this->Resultado;
        }
        else {
                echo "Ocorreu um erro na execução da SQL";
                echo "Erro :" . mysql_error();
                echo "SQL: " . $query;
                die();
                $this->Desconecta();
        }
}
//Esta função desconecta do Banco
public function Desconecta() {
```

```
                return mysql_close($this->Link);
        }

        public function IncluirObservador($Observador) {
                $this->ArrayObservadores[strlen($this->ArrayObservadores)+1] = $Observador;
        }

        public function RemoverObservador($Observador) {

        }

        public function NotificarObservadores() {
                foreach($this->ArrayObservadores as $Observador) {
                        $Observador->Update($this);
                }
        }
}
```

#### *4.1.3.3. O Padrão Template Method*

O Padrão *Template Method*, segundo Ricarte (2006), isola partes fixas das partes variáveis de um algoritmo. FREEMAN E FREEMAN (2007), afirmam que o padrão *Template Method* define o esqueleto de um algoritmo dentro de um método, transferindo alguns de seus passos para as subclasses, permitindo que estas subclasses redefinam certos passos de um algoritmo sem alterar a estrutura do próprio algoritmo. O padrão *Template Method* permite definirmos uma sequência de operações obrigatórias para executar certa tarefa, implementando o que não varia e deixando que cada subclasse implemente sua própria versão da parte que varia. A definição desta sequência de operações, ou algoritmo, é encapsulada em um método, normalmente *Final*, para que as subclasses apenas implementem o que varia, não podendo modificar o algoritmo. Na figura 20 pode ser observada a estrutura do padrão *Template Method*.

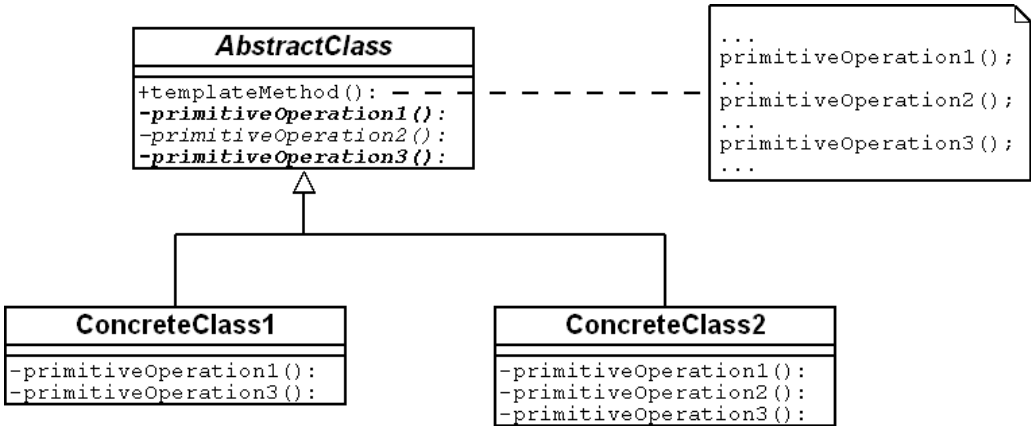


Figura 20: estrutura do Padrão *Template Method*

Fonte: Padrões de Projeto. Ivan Ricarte

Na aplicação exemplo, o padrão *Template Method* foi utilizado para definir o algoritmo de classificação de candidatos nos eventos cadastrados no sistema. O *Template Method* foi implementado na classe abstrata AbstractEvento através do método *Final* FazerClassificacao. Este método define que para fazer a classificação é necessário carregar os cursos, carregar os candidatos e fazer a classificação dos candidatos em cada curso. Na classe abstrata AbstractEvento foram implementados os métodos CarregarCursos e CarregarCandidatos, que não variam de acordo com o tipo de evento. O método ClassificarCandidatos foi definido como abstrato para que cada evento o implemente de acordo com suas regras de classificação. Assim um evento Vestibular irá implementar este método de acordo com as regras de classificação de candidatos em vestibulares e um evento PosGraduação irá implementar este método de acordo com as regras de classificação de candidatos em Pós Graduações. Na figura 21 podem ser observadas as classes que utilizam este padrão.

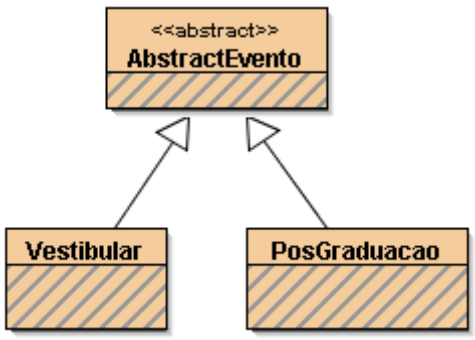


Figura 21: Classes participantes do padrão *Template Method* na aplicação exemplo

Abaixo pode ser observado a implementação do padrão *Template Method* na aplicação exemplo:

```
abstract class Evento {

    ...

    final function FazerClassificacao() {

            $this->CarregarCursos();
            $this->CarregarCandidatos();
            $this->ClassificarCandidatos();
    }

    public function CarregarCursos() {
            echo "Carregando Cursos ...<br>";
    }

    public function CarregarCandidatos() {
            echo "Carregando Candidatos ...<br>";
    }

    abstract function ClassificarCandidatos();

}

class VestibularMG extends Evento{

...

    public function ClassificarCandidatos() {
            echo "Classificando candidatos de acordo com as regras do vestibular ...<br>";
    }
…

}

class PosGraduacao extends Evento{

…
    public function ClassificarCandidatos() {
            echo "Classificando candidatos de acordo com as regras da Pós Graduação ...<br>";
    }
}
```

# 5. CONCLUSÃO

A linguagem PHP, embora não seja uma linguagem orientada a objetos, já possui um excelente suporte a este paradigma que deve ser considerado por todos os desenvolvedores na execução de projetos de softwares também para o ambiente WEB. Na execução de projetos orientados a objetos, o uso de padrões de projeto auxiliam no desenvolvimento de sistemas mais extensíveis e de manutenção mais fácil, já que propicia soluções testadas e comprovadamente eficientes para a resolução de problemas comuns que surgem durante o desenvolvimento. Com o atual suporte à orientação a objetos que o PHP proporciona, é possível utilizar os padrões de projeto para criar aplicações WEB mais profissionais, tornando o PHP não apenas uma linguagem para desenvolvimento de simples *scripts* para tratar dados de formulários mas uma linguagem profissional capaz de ser utilizada para o desenvolvimento de grandes aplicações corporativas.

## 5.1. Contribuições da Monografia

Esta monografia apresenta exemplos práticos da utilização de alguns padrões de projeto com a linguagem PHP, demonstrando que é possível construir aplicações profissionais com esta popular linguagem. Este trabalho também pode orientar desenvolvedores que utilizam a programação estruturada a migrarem para a orientação a objetos, podendo assim obter as vantagens que este paradigma proporciona.

## 5.2. Trabalhos Futuros

Sugerem-se como trabalhos futuros:

a) O desenvolvimento de exemplos dos padrões não citados neste trabalho.

b) Uso de ferramentas de desenvolvimento que tenham os padrões como *template*.

c) Desenvolvimento de um caso único completo, usando diversos padrões simultaneamente.

# REFERÊNCIAS

ACHOUR, Mehdi; BETZ, Friedhelm; DOVGAL, Antony; LOPES, Nuno; OLSON, Philip; RICHTER, Georg; SEGUY, Damien;VRANA, Jakub. *Manual do PHP*. Março 2005. Disponível em < http://www.itmnetworks.com.br/suporte/manual_php.php> Acessado em 12 out 2006.

ALECRIM, Alecrim. *Linguagem ASP*. Set 2005. Disponível em < http://www.infowester.com/lingasp.php.> acessado em mar 2009.

BUYENS, Jim. *Aprendendo MySQL e PHP*. 1.ed. São Paulo: Makron Books, 2002.

CALSAVARA, Alcides. *Padrões de Projeto*. 2009. Disponível em < http://www.ppgia.pucpr.br/~alcides/Teaching/oo/patterns/padroes.ppt > acessado em 05 mar 2009.

DESCHAMPS, Fernando. *Padrões de Projeto:* Uma Introdução. 2009. Disponível em < http://s2i.das.ufsc.br/tikiwiki/apresentacoes/padroes_de_projeto.pdf > acessado em 15 fev 2009.

FERNANDES, Jorge H. C. *Padrões de Design Orientados a Objetos*. 2003. disponível em < http://www.cic.unb.br/~jhcf/MyBooks/iess/Patterns/PatternsIntro-25slides.pdf > acessado em 03 mar 2009.

FERNANDES, Raphaela Galhardo. *Criando e entendendo o Primeiro Servlet*. Fev 2008. Disponível em < ftp://users.dca.ufrn.br/JEEBrasil/IntroducaoServlet.pdf > acessado em 19 mar 2009.

FOWLER, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

FREEMAN, Eric; FREEMAN, Elisabeth. *Padrões de Projeto*. 2. ed. Rio de Janeiro: Alta Books, 2007.

LEITE, Mario; JUNIOR RAHAL, Nelson Abu Sanra. *Programação Orientada ao Objeto:* uma abordagem didática. Mário Leite. Disponível em < http://www.ccuec.unicamp.br/revista/infotec/artigos/leite_rahal.html > acessado em 12 mar 2009.

LEMOS, Tiago. *Programação Orientada a Objeto***:** Herança, Polimorfismo e Encapsulamento . 2009. Disponível em < http://www.tiagolemos.com.br/2009/03/03/programacao-orientada-a-objeto-heranca-polimorfismo-e-encapsulamento/ > acessado em 12 mar 2009.

LOPES, Camilo. *Entendendo Servlets*. Jun 2008. Disponível em < http://camilolopes.wordpress.com/2008/06/09/entendendo-servlet/ > acessado em 19 mar 2009.

MICROSOFT. *ASP.NET*. 2009. Disponível em < http://msdn.microsoft.com/pt-br/asp.net/default.aspx > acessado em 17 mar 2009.

NELSON, Maria Augusta Vieira; NELSON, Torsten Paul. *Padrões de Projeto*. 2009. Disponível em < http://www.noginfo.com.br/arquivos/CC_TEC_08.pdf > acessado em 05 mar 2009.

RICARTE, Ivan Luiz Marques. *Diagrama de Classes UML*. Disponível em <http://www.dca.fee.unicamp.br/cursos/PooJava/desenvolvimento/umlclass.htm > acessado em 12 mar 2009.

RICARTE, Ivan Luiz Marques. *Programação Orientada a Objetos:* Uma Abordagem com Java. 2001. Disponível em < http://www.dep.ufmg.br/professores/miranda/EPD030/ApostilaPOOjava.pdf > acessado em 03 mar 2009.

SANTANA FILHO, Ozeas Vieira; ZARA, Pedro Marcelo. *Microsoft .NET:* uma visão geral para programadores. Senac, 2002. Disponível em < http://books.google.com.br/books?id=RkcbjVKRpX4C&pg=PA93&source=gbs_selected_pages&cad=0_1 > acessado em 17 mar 2009.

WIKIPEDIA. *ASP*. Disponível em < http://pt.wikipedia.org/wiki/ASP > acessado em 12 mar 2009.

WIKIPEDIA. Padrões de Projeto de Software. 2009. Disponível em < http://pt.wikipedia.org/wiki/Padr%C3%B5es_de_projeto > acessado em 25 fev 2009.

ZANDSTRA. Matt. *Entendendo e Dominando o PHP*. 1. ed. São Paulo: Digerati Books, 2006.