PHP5 Orientação a Objetos e Design Patterns

Pablo Dall'Oglio <pablo@dalloglio.net>

O PHP5 traz diversas novidades em termos de recursos, principalmente em relação à Orientação a Objetos. Neste artigo iremos ver de forma sucinta quais são estes recursos e de que forma podemos aplicá-los no dia-a-dia. Vale a pena ressaltar que a maioria dos códigos escritos em PHP4 continuarão funcionando da mesma forma em PHP5, tudo em nome da compatibilidade entre as versões.

Como todos sabem, o PHP é uma linguagem de Script consistente e flexível, com diversas extensões, performance superior e uma grande comunidade. Criada por Rasmus Lerdorf em 1995. Apartir de 1997 o *core* do PHP foi reescrito por Zeev Suravski e Andi Gutmans, formando o que conhecemos hoje por Zend Engine. Na Versão 5 do PHP o *core* Zend Engine atinge a versão 2.0, possibilitando novos recursos que veremos neste artigo.

PS: Este artigo é destinado a quem já possui conhecimentos razoáveis em orientação a objetos. Não entrando em muitos detalhes conceituais.

Passagem de parâmetros

No exemplo a seguir, criaremos uma classe chamada Automovel com 2 métodos, setNome (para alterar o nome do automóvel) e getNome (para obter o nome do automóvel) e uma função chamada alterar (que recebe um objeto do tipo Automovel como parâmetro) e cuja função é alterar o nome do Automóvel para 'Gol'.

```
<?
# Classe Automóvel
class Automovel
{
    # Altera o nome do automóvel
    function setNome($valor)
    {
        $this->nome = $valor;
    }

    # Obtém o nome do automóvel
    function getNome()
    {
        return $this->nome;
    }
}

# Função que altera o nome do
# automóvel para 'Gol'
function alterar($objeto)
{
    $objeto->setNome('Gol');
}

# Instancia novo objeto Automovel
$meucarro = new Automovel;
# Define o Nome para 'Palio'
$meucarro->setNome('Palio');

# Executa função para alterar
# o nome, passando o objeto
# como parâmetro
alterar($meucarro);

# Imprime o nome do Automovel
# na tela.
print $meucarro->getNome();
?>
```

O Objeto é inicializado com o nome 'Pálio', logo em seguida é chamada a função alterar($meucarro), que altera o nome do carro para 'Gol'. O PHP4, não passaria a referência do objeto como parâmetro (*by reference*). Assim, o que era passado para a função era uma cópia do objeto (*by value*)... A função “alterar” trabalhava sobre uma cópia do objeto e externamente à função nada acontecia, como resultado deste programa teríamos:

Palio

A não ser que utilizássemos o operador “&” na frente do parâmetro da função, forçando a passagem de parâmetro *by reference* e obtendo como resultado da função o seguinte:

Gol

Já no PHP5, por padrão, o objeto é passado por referência para a função. Assim, ao executar o código acima no PHP5, obteríamos o seguinte resultado:

Gol

Fábrica de Objetos

Outro conceito importante da orientação a objetos é o Design Pattern *Factory*, também conhecido como Fábrica de Objetos. Uma classe que centraliza a criação dos objetos, o que facilita e padroniza a manutenção do código. Sempre que mudar a política de criação de objetos, basta alterar em somente uma classe. Para tal, criamos uma classe central, responsável pela criação dos objetos, como no exemplo abaixo. Neste exemplo, todas classes estão juntas, mas é recomendado sempre colocar cada classe em um arquivo separadamente, o que torna o sistema mais claro, modular e fácil de ser gerenciado.

```
<?
# Classe Cliente
Class Cliente
{
    // método construtor
    function Cliente()
    {
        echo "criando cliente...\n";
    }
}

# Classe Fornecedor
class Fornecedor
{
    // método construtor
    function Fornecedor()
    {
        echo "criando fornecedor...\n";
    }
}

# Classe Fábrica de Objetos
Class Factory
{
    // Método para criar objetos
    // da classe Cliente
    function CriarCliente($nome)
    {
        return new Cliente($nome);
    }

    // Método para criar objetos
    // da classe Fornecedor
    function CriarFornecedor($nome)
    {
        return new Fornecedor($nome);
    }
}
# Cria objeto Factory
$Fabrica = new Factory;

# Cria instancias de objetos.
$Joao = $Fabrica->CriarCliente('joao');
$Jose = $Fabrica->CriarFornecedor('jose');

# Exibe resultado
var_dump($Joao);
var_dump($Jose);
?>
```

```
Resultado:
criando cliente...
criando fornecedor...
object(Cliente)#2 (0) {
}
object(Fornecedor)#3 (0) {
}
```

No PHP4, a não ser que utilizemos o operador & na frente de 'new Cliente', (o que pode ser confuso para muitos programadores), o PHP iria criar 2 instâncias da clase na memória, uma interna ao método *new* e outra externa. No PHP5, é retornada uma única instância da mesma, facilitando a implementação deste tipo de padrão.

Chamada de métodos sobrecarregados

O PHP4 não permitia “overloaded method calls” ou chamada de métodos sobrecarregados. A forma de resolver isto era decompor o problema em mais linhas como nos exemplos abaixo:

```
<?
$tmp = &$obj->metodo();        // ex: $box = &$botao->get_parent();
$tmp = &$tmp->metodo();        // ex: $janela = &$box->get_parent();
$tmp->metodo();                // ex: $janela->set_title('titulo');
?>
```

O PHP5 irá permitir fazer chamadas de diversos níveis, sobre objetos resultantes ou métodos, como exemplo abaixo. Este recurso será de extrema importância para vários projetos, principalmente em projetos escritos em PHP-GTK:

```
<?
$obj->metodo()->metodo();
func()->metodo()->metodo();
$obj->metodo()->member->metodo();

//exemplo

$botao->get_parent()->get_parent()->set_title('titulo');
?>
```

Abaixo veremos mais um exemplo que só é possível no PHP5, criaremos um objeto e logo em seguida executaremos um método sobre a instância criada

```
CriaPessoa('João')->GetNome();
```

```
<?
# Classe Pessoa
class Pessoa
{
    // Método construtor
    function Pessoa($nome)
    {
        $this->nome = $nome;
    }
    // Método que retorna o nome
```

```
    }
    // Método que retorna o nome
    function GetNome()
    {
        return $this->nome;
    }
}

// Função para criar pessoas
function CriaPessoa($nome)
{
    // Retorna uma instância
    return new Pessoa($nome);
}

// Cria um objeto e imprime o nome
echo CriaPessoa('João')->GetNome();
?>
```

Se estivéssemos utilizando PHP4, seria necessário decompor a linha:

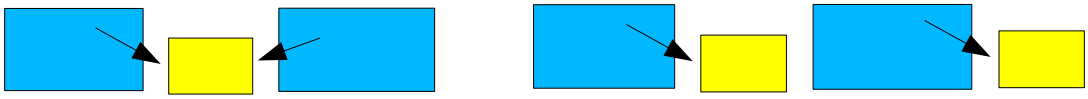
```
echo CriaPessoa('João')->GetNome();
```

em duas linhas:

```
$joao = CriaPessoa('João');
echo $joao->GetNome();
```

Clonagem de objetos

A clonagem de objetos no PHP4 era feita bit a bit. O objeto resultante era exatamente igual ao objeto original. Esse comportamento nem sempre é o desejável. Por exemplo: ao duplicar uma janela em PHP-GTK teremos 2 janelas com o mesmo ID, o que em um ambiente de janelas não é concebível. Outra questão é a seguinte: se o objeto original tiver um objeto filho... o resultante deverá apontar para o mesmo objeto filho ou o objeto filho deverá ser duplicado também ? Veja no gráfico abaixo, os objetos pai estão em azul e os filhos em amarelo. Qual comportamento adotar ?



Para estas e outras questões, uma classe poderá ter um método chamado __clone() responsável por definir o comportamento do objeto ao ser clonado. Para executar a "clonagem", basta declarar um método __clone() na classe que poderá ser clonada. Este método irá definir o comportamento do objeto resultante da clonagem.

No exemplo a seguir, criamos uma classe qualquer chamada Teste. Criamos um objeto $a, instância desta classe. O objeto $a, possui um atributo chamado "atributo", cujo valor é 3. Ao ser clonado, o método __clone() define que o objeto resultante terá o "atributo" contendo o valor 5. Podemos verificar isto facilmente analisando a saída do código seguinte, e também ao comparar os objetos.

```php
class Teste
{
    // Seu método construtor
    function Teste()
    {
        $this->atributo=3;
    }
    // Método para clonagem
    function __clone()
    {
        echo "clonando objeto...\n";
        $this->atributo=5;
    }
}

# Instancia a classe.
$a = new Teste;

# Clona o objeto
$b = clone $a;

# Imprime os atributos de cada objeto.
echo 'atributo de $a ' . $a->atributo . "\n";
echo 'atributo de $b ' . $b->atributo . "\n";


// Verifica se objetos são iguais.
if ($a === $b)
{
    echo 'objeto $a é igual ao objeto $b...';
}
else
{
    echo 'objeto $a é diferente do objeto $b...';
}
?>
```

resultado:

```
clonando objeto...
atributo de $a 3
atributo de $b 5
objeto $a é diferente do objeto $b...
```

Observe que se utilizássemos o comando "**$b = $a**", para clonar o objeto, além deles serem considerados iguais no teste IF() feito no final do programa, os dois teriam o valor do "atributo", como 3.

Métodos construtores/destrutores

No PHP4 não havia mecanismos para destrução de objetos, e para criar o método construtor, era necessário criar um método com o mesmo nome da classe. Agora, no PHP5, é possível definir os métodos __construct() e __destruct(), executados no momento da construção e destruição do objeto. Com isso, é possível realizar um melhor Debug, limpar arquivos temporários, etc...

Para demonstrar tal recurso, utilizaremos o exemplo de uma classe para conexão ao banco de dados. Imagina a seguinte situação: Você abre a conexão, realiza várias consultas e exclui o objeto da memória sem fechar a conexão... Neste caso, o método construtor entra em cena para cuidar exatamente disto. Ele é executado sempre que o objeto é desalocado da memória.

```
<?
/*
 * Classe Conexao
 * Realiza consulta ao Banco de Dados
 */
class Conexao
{
    private $conn;

    // Método construtor
    // Abre conexão ao BD
    function __construct()
    {
        echo "construindo..: ";
        $this->conn = pg_connect('host=localhost dbname=exemplos user=postgres
');
        echo "conexao aberta.\n";
    }

    // Realiza consulta
    function query($sql)
    {
        echo "consultando...\n";
        return pg_query($this->conn, $sql);
    }

    // método destrutor, fecha conexão
    function __destruct()
    {
        echo "destruindo..: ";
        pg_close($this->conn);
        echo "conexao fechada.\n";
    }
}

// instancia classe Conexão
$conn = new Conexao;

// realiza consulta
$result = pg_fetch_all($conn->query("select codigo, descricao from cidade where
ref_estado='RS'"));

// imprime resultados
echo "==================\n";
foreach ($result as $linha)
{
```

```
    echo $linha['codigo'] . '-'. $linha['descricao'] . "\n";
}
echo "====================\n";

// deleta objeto
unset($conn);
echo "END\n";
?>
```

```
construindo..: conexao aberta.
consultando...
====================
1-Porto Alegre
2-Lajeado
3-Cruzeiro do Sul
4-Encantado
====================
destruindo..: conexao fechada.
END
```

Encapsulamento

No PHP5 foram introduzidos os operadores Public, Protected e Private.

- Public: Todos podem acessar membros da classe diretamente.
- Protected: Os membros da classe podem ser acessados de dentro da mesma e apartir de classes filhas.
- Private: Os membros da classe somente podem ser acessados de dentro da mesma.

Ambos propriedades e métodos podem ser do tipo {public, protected, private}. Caso o método ou a propriedade não seja declarado como {public, protected, private}, automaticamente será do tipo public, para manter a compatibilidade com versões anteriores.

Veja o exemplo a seguir onde a variável salario só pode ser acessada da classe Pessoa, ou de suas filhas (Empregado)...

```
<?
class Pessoa
{
   protected $salario;
}

class Empregado extends Pessoa
{
   function GetSalario()
   {
      return $this->salario;
   }
}
?>
```

Assim como podemos encapsular as propriedades, podemos encapsular também métodos. Veja no exemplo abaixo, onde temos o método ObtemIdade() da classe Pessoa, que retorna a idade da pessoa. Este método só poderá ser chamado da própria classe Pessoa ou de uma de suas filhas (como Funcionario). Na classe Funcionario, o método CalculaSalario() utiliza o método

ObtemIdade() da classe pai para retornar o salário. Observe que o método ObtemIdade é do tipo *protected*, assim sendo este código retornará um erro na última linha do programa, pois o mesmo não pode ser invocado externamente à classe.

```
<?
# Classe abstrata Pessoa
class Pessoa
{
    // Método que retorna a idade.
    protected function ObtemIdade()
    {
        return $this->idade;
    }
}

# Classe concreta Funcionario
class Funcionario extends Pessoa
{
    // Método que calcula o salário
    function CalculaSalario()
    {
        return 500 + (Pessoa::ObtemIdade()*20);
    }
}

# Objeto joao, da classe Funcionario
$joao = new Funcionario;
$joao->idade = 20;

# Imprime o Salário
echo $joao->CalculaSalario();

# Imprime a Idade
echo $joao->ObtemIdade();
?>
```

```
resultado:
900
Fatal error: Call to protected method Pessoa::ObtemIdade() from context '' in
heran.php on line 30
```

O Método __set()

Ainda para prover encapsulamento, o PHP5 introduz o método __set(). O método __set() pode ser declarado em qualquer classe e será executado toda vez que for atribuído algum valor à alguma propriedade do objeto. Ou seja, ele intercepta a atribuição de valores à atributos de um objeto.

No exemplo a seguir, temos uma classe Pessoa, e um atributo chamado “nascimento”, que representa a data de nascimento da pessoa. A forma correta de alterar o valor do atributo nascimento, é chamando o método SetNascimento, passando o dia, o mês e o ano como parâmetros. Para evitar que façamos atribuições erradas, como atribuir o valor “04 de março de 2004” ao atributo nascimento, declaramos o método __set(), que fará esse teste de consisência para nós.

```php
<?
# Classe Pessoa
class Pessoa
{
    # Método para interceptar a
    # atribuição de valores
    # Recebe os 2 parâmetros por padrão.
    function __set($propriedade, $valor)
    {
        # Se é a propriedade nascimento
        # que está sendo atribuída
        if ($propriedade == 'nascimento')
        {
            # Testa se o valor a ser atribuído
            # é composto por 3 partes separadas por "-"
            if (count(explode('-', $valor)) == 3)
            {
                # atribui o valor
                $this->$propriedade = $valor;
                # mensagem de OK
                echo "[OK SET]: propriedade: $propriedade, valor: $valor\n";
            }
            else
            {
                # mensagem de ERRO
                echo "[ERRO SET]: propriedade: $propriedade, valor: $valor\n";
            }
        }
    }
    # Método para alterar a data de nascimento
    # no formato correto "dia-mes-ano"
    function SetNascimento($dia, $mes, $ano)
    {
        $this->nascimento = "$dia-$mes-$ano";
    }

    # Método para retornar
    # a data de nascimento
    function GetNascimento()
    {
        return $this->nascimento;
    }
}

# Cria objeto maria
# do tipo Pessoa
$maria = new Pessoa;

# atribui a data de nascimento
# de forma errada.
$maria->nascimento = '04 março de 2004';

# informa o valor do atributo Nascimento
echo "Valor do atributo: " . $maria->GetNascimento() . "\n";

echo "\n";

# atribui a data de nascimento
```

```
# através do método correto.
$maria->SetNascimento(10,10,2004);

# informa o valor do atributo Nascimento
echo "Valor do atributo: " . $maria->GetNascimento() . "\n";
?>
```

```
resultado:
[ERRO SET]: propriedade: nascimento, valor: 04 março de 2004
Valor do atributo:

[OK SET]: propriedade: nascimento, valor: 10-10-2004
Valor do atributo: 10-10-2004
```

Classes Abstratas

No PHP5, é implementado o conceito de classe abstrata. Classe abstrata é uma classe que não pode ter instâncias diretas. Ela só pode ser herdada. Neste exemplo ocorrerá um erro pois tentaremos instanciar a classe Pessoa, que é justamente declarada como abstrata... Neste exemplo o correto seria instanciar a classe Funcionario.

```
<?
# Classe abstrata Pessoa
abstract class Pessoa
{
   // ...
}

# Classe concreta Funcionario
class Funcionario extends Pessoa
{
   // ...
}

# Objeto joao, da classe Pessoa
$joao = new Pessoa;

# Objeto joao, da classe Funcionario
$joao = new Funcionario;

?>
```

```
resultado:
Fatal error: Cannot instantiate abstract class Pessoa in heranca.php on line 15
```

Constantes de Classe

No PHP5 é introduzido também o conceito de constante de classe. É uma constante acessível a qualquer momento e representa uma informação dentro do contexto da classe.

```
<?
class Pessoas
{
    const Numero = 5;
}

echo Pessoas::Numero;
?>
```

```
resultado = 5
```

Interfaces

No PHP5 é introduzido o conceito de interfaces. Uma interface cria um protocolo que define funções que devem ser implementadas por uma classe. Veja o erro gerado no caso abaixo. A Classe Palio implementa a interface Automovel. Como na interface Automovel possui a definição de uma função chamada Ligar(), a classe Palio também deveria tê-la (está comentado).

```
<?php
interface Automovel
{
    public function Ligar();
}

class Palio implements Automovel
{
    /*
    public function Ligar()
    {
    }
    */
}
?>
```

```
Resultado:

Fatal error: Class Palio contains 1 abstract methods and must therefore be
declared abstract (Automovel::Ligar) in inter.php on line 14
```

Propriedades Estáticas

No PHP4 já era possível a chamada de métodos estáticos, ou seja, executar um método sem a necessidade de instanciar uma classe. O PHP5 introduz as propriedades estáticas ou variáveis de classe. Veja no exemplo abaixo... a variável $qtde_instancias é uma propriedade estática, ela existe dentro do contexto da classe Pessoa.

```
<?
# Classe Pessoa
class Pessoa
{
    # Propriedade estática
    # quantidade de instancias
    static $qtde_instancias;

    # Retorna a quantidade de instancias
    function getQtde()
    {
        return Pessoa::$qtde_instancias;
    }
    # imprime a empresa
    static function getEmpresa()
    {
            return 'Linux Corporation';
    }
}
# Imprime a empresa.
print Pessoa::getEmpresa() . "\n";

# Cria uma instancia
$joao = new Pessoa;

# Incrementa a qtde de instancias.
Pessoa::$qtde_instancias ++;

# Exibe a qtde de instancias.
print Pessoa::GetQtde() . "\n";

?>
```

Resultado:

```
Linux Corporation
1
```

Design Pattern Singleton

Propriedades estáticas podem ser utilizadas também para implementar o Design Pattern conhecido como Singleton.

Muitas vezes, ao implementarmos um programa, precisamos que determinado recurso (objeto) seja instanciado somente uma única vez, não importando quantas partes do programa utilizam este recurso, o mesmo não deve ser instanciado diferentes vezes para diferentes utilizações. Um exemplo prático são interfaces de hardware, ou mesmo objetos de conexão com o banco de dados, onde devemos ter um único link de conexão para múltiplas operações, ou mesmo um sistema onde temos várias impressoras conectadas ao sistema, mas apenas uma única fila de impressão. Logo o objetivo do Design Pattern Singleton é garantir que a classe tenha uma única instância e também garantir um ponto de acesso à esta instância.

No Nosso exemplo, teremos uma Classe de Conexão com o Banco de Dados, chamada ConexaoBancoDados, e utilizaremos o conceito de propriedades estáticas e o método RetornaInstancia(), para implementar o Design Pattern Singleton.

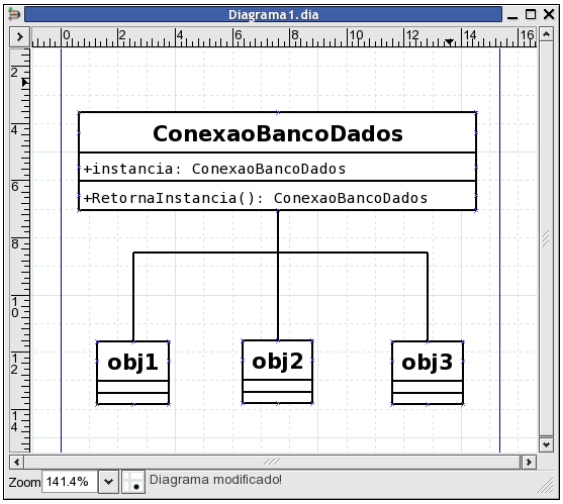


Figura 1 – Design Pattern Singleton

```
<?
# Classe para conexões ao Banco de Dados
class ConexaoBancoDados
{
    // propriedade estática que armazenará
    // a instância única.
    private static $instancia = null;

    // Método construtor qualquer
    private function __construct()
    {
    }

    // Método para obter a instância
    public static function RetornaInstancia()
    {
        // caso não exista a instância
        // instancia [primeira vez]
        if (self::$instancia == null)
        {
            // instancia objeto
            echo "Criando instancia\n";
```

```
            self::$instancia = new ConexaoBancoDados;
        }
        // retorna a instância criada.
        return self::$instancia;
    }
}

// Obtém a instância de Conexão.
$a = ConexaoBancoDados::RetornaInstancia();
// Obtém a instância de Conexão.
$b = ConexaoBancoDados::RetornaInstancia();

// Testa se os objetos são iguais.
if ($a ===$b)
{
    echo '$a e $b são o mesmo objeto';
}
else
{
    echo '$a e $b não são o mesmo objeto';
}
?>
```

resultado:

```
Criando instancia

$a e $b são o mesmo objeto
```

Lazy Initialization e o método __get()

O método __get(), introduzido no PHP5, serve para interceptar o retorno de propriedades de um objeto. Dentre outras funcionalidades, é através dele que podemos implementar o conceito de lazy initialization ou inicialização tardia (ou preguiçosa).

Para entender melhor, utilizaremos um exemplo de agregação. A agregação acontece quando temos um objeto que possui referência a outro(s) objeto(s). Um carro por exemplo, possui diversos objetos agregados, dentre eles: motor, bancos, rodas, correceria, dentre outros. Os objetos agregados "geralmente" são criados no método construtor da classe, permitindo que eles fiquem disponíveis para todos os outros métodos da mesma. Veja no exemplo:

```
<?
# Classe Motor
class Motor
{
    function start()
    {
        echo "ligando...\n";
    }
}
# Classe Carro
class Carro
{
    # Método Construtor
    function Carro()
    {
        $this->Motor = new Motor;
    }
    # Método para ligar.
```

```
    function Liga()
    {
        $this->Motor->start();
    }
}
# Instancia classe Carro.
$meucarro = new Carro;
$meucarro->Liga();

?>
```

resultado:

```
ligando...
```

Neste caso, o objeto agregado motor é criado no método construtor da classe Carro, ficando disponível para qualquer método que venha utilizá-lo. Mas em muitos casos, só utilizaremos o objeto agregado em algumas circunstâncias. Assim sendo, inicializar todos objetos agregados no método construtor pode gerar algum *overhead*.

Para resolver isso, o objeto Motor deveria ser inicializado somente quando necessário. Esta funcionalidade é dada pelo método __get(), que intercepta a requisição das propriedades de uma classe, automaticamente inicializando o objeto agregado quando ele é requisitado pela primeira vez. Veja como o código ficaria utilizando este conceito logo abaixo. Observe que o método __get() recebe por padrão um parâmetro que é o nome do atributo a ser requisitado, então passa a verificar através de um IF() qual a propriedade requisitada, inicializando-a... Veja que o resultado é o mesmo:

```
<?
# Classe Motor
class Motor
{
    function start()
    {
        echo "ligando...\n";
    }
}
# Classe Carro
class Carro
{
    # Método para ligar.
    function Liga()
    {
        $this->Motor->start();
    }

    # Método interceptador
    function __get($propriedade)
    {
        if ($propriedade == 'Motor')
            return new Motor;
    }
}
# Instancia classe Carro.
$meucarro = new Carro;
$meucarro->Liga();
?>
```

resultado:

```
ligando...
```

Delegation e o método __call()

Nesta seção iremos abordar a utilização do conceito chamado de *Delegation*, que significa passar a responsabilidade de execução de uma tarefa de um objeto para outro, por exemplo. O conceito *Delegation* é utilizado para implementar um conhecido Design Pattern, o *Strategy*, que define e encapsula uma família de algoritmos (estratégias) e os torna intecambeáveis.

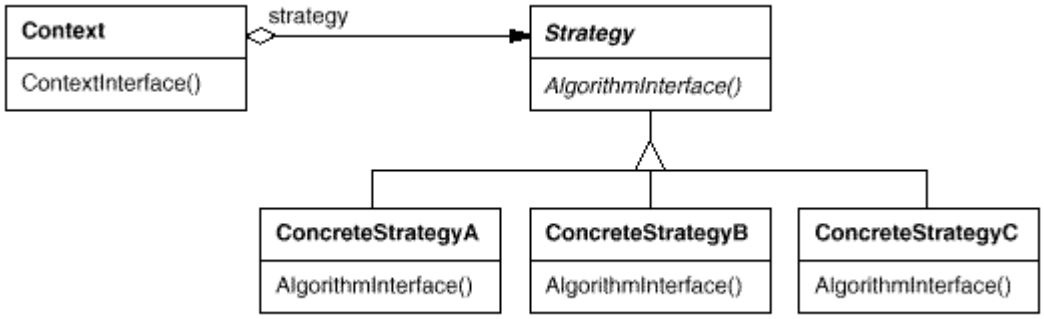


Figura 2 – Design Pattern Strategy

No exemplo que iremos abordar, temos uma classe chamada Arquivo, e diversas outras: ArquivoHtml, ArquivoTexto, etc. A classe Arquivo trata das operações básicas, como Abrir(), Fechar() e Apagar(). As classes individuais para cada tipo de arquivo, como ArquivoHtml, ArquivoTexto, etc, tratam do algoritmo de exportação dos dados naqueles formatos. O método a ser chamado na classe Arquivo, para exportar para os formatos, se chama ExportaComoHtml(), ExportaComoTexto, etc. Como este método não existe na classe Arquivo, o método __call() intercepta a chamada e trata de instanciar as classes necessárias.

```
<?
# Estratégia para exportar como Html
class ArquivoHtml
{
    // ...
}

# Estratégia para exportar como Texto
class ArquivoTexto
{
    // método para exportar como TXT.
    function Exporta($arquivo)
    {
        echo "exportando como TXT: $arquivo\n";
    }
}

# Classe principal para lidar com Arquivos
class Arquivo
{
    // método para Abrir arquivo.
    function Abrir($arquivo)
    {
        echo "opening $arquivo...\n";
    }

    // método para interceptar chamada de métodos
    function __call($metodo, $parametro)
    {
        // teste do método chamado
```

```
        if ($metodo =='ExportaComoTxt')
        {
            // delega a responsabilidade
            // instanciando a estratégia X
            $obj = new ArquivoTexto;
            $obj->Exporta($parametro[0]);
        }
    }
}

// Instancia classe Arquivo
$arquivo = new Arquivo;
// Abre arquivo.
$arquivo->Abrir('teste.dat');
// Exporta como Texto
$arquivo->ExportaComoTxt('teste.txt');
?>

Resultado:
opening teste.dat...
exportando como TXT: teste.txt
```

Se o mesmo programa fosse executado no PHP4, a seguinte mensagem seria executada:

```
Fatal error: Call to undefined function: exportacomotxt() in file.php on line 46
```

Uma idéia que poderia ser implementada é a de agregar dinamicamente objetos à um objeto qualquer. Sempre que for chamado um método que não exista no objeto principal, o método __call() poderia pesquisar nesta pilha de objetos algum que tenha um método que satisfaça a mesma “assinatura” da chamada.

Template Method

Template Method é um Design Pattern que provê a definição de uma estrutura abstrata de classes com operações similares. No Template Method as subclasses podem redefinir o comportamento das classes "sem alterar sua estrutura". As partes invariantes do algoritmo são implementadas na superclasse e as variantes são customizadas nas subclasses. As operações a serem redefinidas nas subclasses podem ser marcadas como obrigatórias (as subclasses devem implementar) ou opcionais. O uso de Template Method é comum em frameworks, onde o framework implementa as partes invariantes e o programador especializa as classes de acordo com sua necessidade. Também é utilizado em conjunto com o Design Pattern *Strategy*, visto anteriormente.

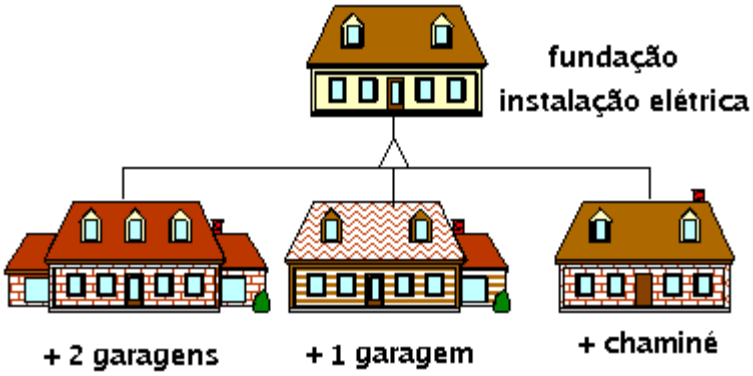


Figura 3 – Design Pattern Template Method

No exemplo abaixo, implementamos uma Classe abstrata Casa, cujo próprio método construtor é o nosso template method. Para que ele não seja sobre-escrito nas subclasses, é necessário utilizar o operador do PHP5, **final**. Para que apenas ELE possa ser chamado das subclasses, é declarado como public, e os demais como protected. Nosso exemplo se concentra na construção de uma casa, onde algumas operações obrigatoriamente precisam estar definidas nas subclasses e outras são opcionais. Para que uma operação seja obrigatória na subclasse, é declarada como abstract, como no caso do método Eletricidade(), que define a instalação elétrica da casa. Já o método Garagem(), que define a existência de garagem na casa, é opcional, podendo não existir na subclasse.

```
<?
# Classe abstrata Casa
abstract class Casa
{
    # Método Construtor
    public final function Casa()
    {
        print "Template::Construção\n";
        // Chama métodos específicos
        $this->Eletricidade();
        $this->Garagem();
    }
    // declara método como obrigatório
    protected abstract function Eletricidade();
    // método opcional
    protected function Garagem() { }
}

# SubClasse MinhaCasa
```

```php
class MinhaCasa extends Casa
{
    // Método Instalação Elétrica
    protected function Eletricidade()
    {
        print "Concreta::Eletricidade\n";
    }

    // Método Constrói Garagem
    protected function Garagem()
    {
        print "Concreta::Garagem\n";
    }
}

# Cria nova instância de MinhaCasa.
$a = new MinhaCasa;
?>
```

Resultado:

```
Template::Construção
Concreta::Eletricidade
Concreta::Garagem
```

Caso seja por engano redeclarado o template method na subclasse, o seguinte erro ocorrerá:

```
Fatal error: Cannot override final method Casa::Casa()
```

Caso o método obrigatório Eletricidade() não seja declarado na subclasse, a seguinte mensagem será exibida:

```
Fatal error: Cannot instantiate abstract class MinhaCasa
```

Referências

- Changes in PHP 5/Zend Engine 2.0 by PHP Group.
- Introduction to Interceptors II: Implementing Lazy Initialization por Sebastian Bergmann
- Introduction to Interceptors I: Implementing Delegation por Sebastian Bergmann
- Design Patterns: Elements of Reusable Object-Oriented Software. Gamma *et. al*.
- The Singleton Design Pattern by Brian d foy.

Pablo Dall'Oglio começou sua carreira como desenvolvedor clipper em 1995 quando desenvolveu uma série de sistemas para automação de escolas e de empresas. Desde 2000, trabalha somente com a linguagem PHP. É autor de projetos em software livre conhecidos como o GNUTeca (www.gnuteca.org.br), Agata Report (http://agata.dalloglio.net) e do editor Tulip (http://tulip.dalloglio.net). Entusiasta de PHP-GTK desde sua criação em 2001, se tornou amigo de seu criador, Andrei Zmievski. Émembro do time de documentação e tradução do PHP-GTK. É autor do único livro sobre PHP-GTK no mundo (http://www.php-gtk.com.br/), tem diversos artigos publicados em revistas nacionais e internacionais sobre PHP, GTK, WebServices e PostgreSQL. É especialista em Modelagem de Bancos de Dados, Engenharia de software, orientação a objetos, PostgreSQL, PHP e PHP-GTK. Pode ser contactado pelo e-mail pablo@php.net ou pablo@dalloglio.net.

Maio de 2004