Design Patterns (Padrões de Projeto)

Existem diversas formas de estruturar o código e o projeto da sua aplicação web e você pode gastar muito ou pouco esforço pensando na sua arquitetura. Mas geralmente é uma boa ideia seguir à padrões comuns, pois isso irá fazer com que seu código seja mais fácil de manter e de ser entendido por outros desenvolvedores.

- Padrões de Arquitetura na Wikipedia
- Padrões de Design de Software na Wikipedia
- Coleção de exemplos de implementação

Factory (Fábrica)

Um dos padrões de design mais utilizados é o padrão "Factory" (Fábrica). Através dele uma classe simplesmente cria o objeto que você gostaria de usar. Considere o seguinte exemplo desse padrão de design:

```
<?php
class Automobile
{
    private $vehicle_make;
    private $vehicle_model;

    public function __construct($make, $model)
    {
        $this->vehicle_make = $make;
        $this->vehicle_model = $model;
    }

    public function get_make_and_model()
    {
        return $this->vehicle_make . ' ' . $this->vehicle_model;
    }
}

class AutomobileFactory
{
    public static function create($make, $model)
    {
        return new Automobile($make, $model);
    }
}

// Solicita a "Factory" que crie o objeto Automobile
$veyron = AutomobileFactory::create('Bugatti', 'Veyron');

print_r($veyron->get_make_and_model()); // imprime "Bugatti Veyron"
```

Esse código usa uma "Factory" para criar o objeto do tipo "Automobile". Existem dois possíveis benefícios para criar seu código dessa forma, o primeiro é que se você precisar mudar, renomear ou substituir a classe Automobile futuramente você pode fazer e só terá que modificar o código na "Factory", em vez de em todos os lugares do seu projeto onde você usa a classe Automobile. O segundo benefício possível é que caso a criação do objeto seja um processo complicado, você pode executar todo esse trabalho na factory, em vez de repetí-lo toda vez que precisar criar uma nova instância da classe.

Usar o padrão de “Factory” não é sempre necessário (ou esperto). O código de exemplo usado aqui é tão simples que essa “Factory” estaria simplesmente adicionando complexidade indesejada. Entretanto se você estiver realizando um projeto um pouco maior ou mais complexo você pode se salvar de muitos problemas com o uso do padrão “Factory”.

- Padrão “Factory” na Wikipedia

Singleton (Única Instancia)

Quando arquitetando uma aplicação web, é comum fazer sentido tanto conceitualmente quanto arquitetonicamente permitir o acesso a somente uma instância de uma classe em particular, o padrão “Singleton” nos permite realizar essa tarefa.

```
<?php
class Singleton
{
    /**
     * Retorna uma instância única de uma classe.
     *
     * @staticvar Singleton $instance A instância única dessa classe.
     *
     * @return Singleton A Instância única.
     */
    public static function getInstance()
    {
        static $instance = null;
        if (null === $instance) {
            $instance = new static();
        }

        return $instance;
    }

    /**
     * Construtor do tipo protegido previne que uma nova instância da
     * Classe seja criada através do operador `new` de fora dessa classe.
     */
    protected function __construct()
    {
    }

    /**
     * Método clone do tipo privado previne a clonagem dessa instância
     * da classe
     *
     * @return void
     */
    private function __clone()
    {
    }

    /**
     * Método unserialize do tipo privado para prevenir a desserialização
     * da instância dessa classe.
     *
     * @return void
     */
    private function __wakeup()
    {
    }
```

```
}

class SingletonChild extends Singleton
{
}

$obj = Singleton::getInstance();
var_dump($obj === Singleton::getInstance());             // bool(true)

$anotherObj = SingletonChild::getInstance();
var_dump($anotherObj === Singleton::getInstance());      // bool(false)

var_dump($anotherObj === SingletonChild::getInstance()); // bool(true)
```

O código acima implementa o padrão “Singleton” usando uma variável *estática* e o método estático de criação `getInstance()`. Note o seguinte:

- O construtor `__construct` é declarado como protegido para prevenir que uma nova instância seja criada fora dessa classe pelo operador `new`.
- O método mágico `__clone` é declarado como privado para prevenir a clonagem dessa instância da classe pelo operador `clone`.
- O método mágico `__wakeup` é declarado como privado para prevenir a desserialização de uma instância dessa classe pela função global `unserialize()`.
- Uma nova instância é criada via late static binding no método de criação `getInstance()` declarado como estático. Isso permite a criação de classes “filhas” da classe Singleton no exemplo

O padrão Singleton é útil quando você precisa garantir que somente uma instância da classe seja criada em todo o ciclo de vida da requisição em uma aplicação web. Isso tipicamente ocorre quando você tem objetos globais (tais como uma classe de Configuração) ou um recurso compartilhado (como uma lista de eventos).

Você deve ser cauteloso quando for usar o padrão “Singleton” já que pela sua própria natureza ele introduz um estado global na sua aplicação reduzindo a possibilidade de realização de testes. Na maioria dos casos Injeção de Dependências pode (e deve) ser usado no lugar de uma classe do tipo Singleton. Usar Injeção de Dependências significa não introduzir acoplamento desnecessário no design da sua aplicação, já que o objeto usando o recurso global ou compartilhado não necessita de conhecimento sobre uma classe concretamente definida.

- Padrão Singleton na Wikipedia

Strategy (Estratégia)

Com o padrão “Strategy” (Estratégia) voce encapsula famílias específicas de algoritimos permitindo com que a classe cliente responsável por instanciar esse algoritimo em particular não necessite de conhecimento sobre sua implementação atual. Existem várias variações do padrão “Strategy” o mais simples deles é apresentado abaixo:

O primeiro bloco de código apresenta uma familia de algorítimos; você pode querer uma array serializado, um JSON ou talvez somente um array de dados:

```
<?php
```

```
interface OutputInterface
{
    public function load();
}

class SerializedArrayOutput implements OutputInterface
{
    public function load()
    {
        return serialize($arrayOfData);
    }
}

class JsonStringOutput implements OutputInterface
{
    public function load()
    {
        return json_encode($arrayOfData);
    }
}

class ArrayOutput implements OutputInterface
{
    public function load()
    {
        return $arrayOfData;
    }
}
```

Através do encapsulamento do algoritimo acima você está fazendo seu código de forma limpa e clara para que outros desenvolvedores possam facilmente adicionar novos tipos de saída sem que isso afete o código cliente.

Você pode ver como cada classe concreta ‘output’ implementa a OutputInterface - isso serve a dois propósitos, primeiramente isso prevê um simples contrato que precisa ser obedecido por cada implementação concreta. Segundo, através da implementação de uma interface comum você verá na próxima seção que você pode utilizar Indução de Tipo para garantir que o cliente que está utilizando esse comportamento é do tipo correto, nesse caso ‘OutputInterface’.

O próximo bloco de código demonstra como uma classe cliente relizando uma chamada deve usar um desses algorítimos e ainda melhor definir o comportamento necessário em tempo de execução:

```
<?php
class SomeClient
{
    private $output;

    public function setOutput(OutputInterface $outputType)
    {
        $this->output = $outputType;
    }

    public function loadOutput()
    {
        return $this->output->load();
    }
}
```

A classe cliente tem uma propriedade private que deve ser definida em tempo de execução e ser do tipo ‘OutputInterface’ uma vez que essa propriedade é definida uma chamada a loadOutput() irá chamar o método load() na classe concreta do tipo ‘output’ que foi definida.

```
<?php
$client = new SomeClient();

// Quer um array?
$client->setOutput(new ArrayOutput());
$data = $client->loadOutput();

// Quer um JSON?
$client->setOutput(new JsonStringOutput());
$data = $client->loadOutput();
```

- Padrão “Strategy” na Wikipedia

Front Controller

O padrão front controller é quando você tem um único ponto de entrada para sua aplicação web (ex. index.php) que trata de todas as requisições. Esse código é responsável por carregar todas as dependências, processar a requisição e enviar a resposta para o navegador. O padrão Front Controller pode ser benéfico pois ele encoraja o desenvolvimento de um código modular e provê um ponto central no código para inserir funcionalidades que deverão ser executadas em todas as requisições (como para higienização de entradas).

- Padrão Front Controller na Wikipedia

Model-View-Controller

O padrão model-view-controller (MVC) e os demais padrões relacionados como HMVC and MVVM permitem que você separe o código em diferentes objetos lógicos que servem para tarefas bastante específicas. Models (Modelos) servem como uma camada de acesso aos dados onde esses dados são requisitados e retornados em formatos nos quais possam ser usados no decorrer de sua aplicação. Controllers (Controladores) tratam as requisições, processam os dados retornados dos Models e carregam as views (Visões) para enviar a resposta. E as views são templates de saída (marcação, xml, etc) que são enviadas como resposta ao navegador.

O MVC é o padrão arquitetônico mais comumente utilizado nos populares Frameworks PHP.

Leia mais sobre o padrão MVC e os demais padrões relacionados:

- MVC
- HMVC
- MVVM

Quick Tips: Padrões de Projeto no PHP

Motivação

Olá pessoal, vamos ver nesta tips um pouco sobre padrões de projeto no PHP.

Sem dúvida o que não falta hoje são opções de linguagens orientadas a objetos para desenvolvermos nossos projetos, seja para desktop ou mesmo para Web o fato e que não são raras as situações que os programadores se encontram. Eles acabam sendo vitimas de suas próprias armadilhas, usam tanto a herança que ao se modificar um simples método na classe cliente, por exemplo, todo o modelo para de funcionar, ou pelo menos não funciona mais como deveria. Isso se deve ao fato do projeto está totalmente acoplado, os mais otimistas dizem “Integrado”. Porem se repararmos em todos os projetos que vamos desenvolver passa pelos mesmos problemas.

Na década de 70 um engenheiro civil chamado Cristopher Alexander começou e reparar que embora muito diferentes, as construções daquela época seguiam um padrão de arquitetura, da menor a mais sofisticada, todas passavam pelos mesmos problemas e que as soluções para estes problemas eram quase sempre as mesmas.

Foi ai que ele decidiu catalogar esses problemas e apresentar para cada um deles uma solução comum. Daí surgiu a bíblia da Engenharia Civil (The Timeless Way of Building) que tratava exatamente de padrões para construção na área da engenharia civil.

Segundo Christopher Alexander, "Cada Pattern descreve um problema o qual ocorre repetidamente em nosso ambiente, e então descreve um conjunto de soluções para este problema, de maneira que você possa usar esta solução um milhão de vezes, sem o fazer da mesma maneira duas vezes."

Foi ai que tudo começou, pois foi a partir deste trabalho que Eric Gama e seus amigos decidiram escrever padrões de projetos porem aplicados ao desenvolvimento softwares orientados a objetos e em 1995 lançaram a bíblia do desenvolvimento de software *Design Patterns: Elements of Reusable Object-Oriented Software” - 1995* que veio a ser o ponto de referencia para todo programador que desenvolve OO.

Os padrões do GOF (Gang of Four), como são conhecidos, são divididos em três categorias: Criacionais, Estruturais e Comportamentais.

Criacionais

Padrões criacionais abstraem o processo de instanciação. Ajudam a criar um sistema independente de como o objeto é criado, composto e representado. Exemplo:

- Singleton;

- Propotype;
- Factory Method;
- Abstract Factory;
- Builder.

Estruturais

Padrões estruturais definem como as classes e os objetos são compostos para dar forma a estruturas maiores. Exemplo:

- Adapter;
- Composite;
- Façade;
- Proxy.

Comportamentais

Padrões comportamentais definem a maneira pela quais iremos interagir com as classes e objetos e como eles interagem entre si.

Exemplo:

- Iterator;
- Observer;
- State;
- Strategy;
- Template Method.

Os padrões GOF são no total de 23, aqui eu destaquei apenas os principais.

Esses padrões não pertencem a esta ou aquela linguagem, eles são na verdade boas praticas de desenvolvimento de softwares que se utilizadas nos garantem que evitaremos uma serie de problemas comuns no desenvolvimento de software. Portanto para utilizarmos no PHP devemos apenas seguir essas padrões no desenvolvimento de nossas classes e Objetos. E lembre-se um patterns é classificado segundo o problema que ele resolve e não quanto ao local em que e implementado.

A seguir veremos com mais detalhe dois tipos de padrão.

Singleton

Há caso em que se faz necessário garantir que uma e apenas uma instância de uma classe possa estar em uso no nosso modelo, um bom exemplo seria uma classe usuario, para que ter dois objetos quando um já seria suficiente. Este é um problema e para eles temos um pattern então pra que reinventar a roda? Observe:

```
class Usuario{
        static private $instance = null;
        private $nome;
        private $senha;
        private function __construct(){

        }

        static function getinstance(){
               if (self::$instance == null){
                   self::$instance = new Usuario;
               }
                   return self::$instance;
        }
```

Veja que o construtor da classe usuário e private e isso tem um motivo claro. Não podemos permitir que um novo usuário seja criado pelo método new e sim através da função estática getinstance e esta se encarrega de criar um novo objeto ou retorna um já existente. Com isso garantirmos que um e somente um usuário estará instanciado por vez.

Factoty Method

Em outros casos precisamos concentrar a maneira como os nossos objetos são criados em um mesmo lugar e com isso se uma nova regra surgir em relação a criação desses objetos alteramos em apenas um local. Para isso criamos uma classe central que fica responsável por criar nossos objetos.

```
interface ICarro{
   function Ligar();
}

class Gol implements ICarro{
      function Ligar(){echo "Gol Ligado !!!!"; }
}

class Palio implements ICarro{
      function Ligar(){echo "Palio Ligado !!!!"; }
}
```

Temos ai duas classes que implementam a mesma interface que possui um método Ligar() que e implementado de maneira pertinente a cada classe. Agora criaremos uma classe que ficara encarregada de criar nossos objetos:

```
class FactoryCar{
   static function CriarCarro($marca){
     switch ($marca){
        case "Gol":  $obj = new Gol;
                          $obj->Ligar();
        break;
        case "Palio": $obj = new Palio;
                           $obj->Ligar();
     }
   }
}
```

Basta agora fazermos uso desta ultima classe para instanciarmos nossos objetos:

```
FactoryCar::CriarCarro(“Palio”);  ou
FactoryCar::CriarCarro(“Gol”);
```

Sem duvida programar com PHP 5 é muito mais eficiente do que em outras versões. Temos disponíveis hoje recursos que nos permitem criar aplicações profissionais. Lembre-se Orientação a Objetos não é uma característica desta ou daquela linguagem é uma filosofia de desenvolvimento que aliada a padrões nos permitem atingir a excelência em nossos projeto.

https://www.devmedia.com.br/quick-tips-padroes-de-projeto-no-php/14452

Padrões de projeto (Design Patterns) no PHP

Nesta série de artigos quero falar um pouco sobre padrões de projeto e alguns assuntos além, vou começar abordando as patterns listadas pela GoF, obviamente não teremos um projeto prático, será mais ou menos um manual sobre o assunto.

Tem um curso introdutório (e grátis) sobre o padrões de projeto aqui no WebDevBr.

O que é padrão de projeto ou design pattern

Padrão de projeto é a tradução de Design Pattern (???, eu deveria colocar "o mesmo que" no lugar de a "tradução de"?), e tem como objetivo resolver problemas que ocorrem com frequência dentro da orientação a objetos.

A Wikipedia ainda define:

> Em engenharia de software, um padrão de desenho (português europeu) ou padrão de projeto (português brasileiro) (do inglês design pattern) é uma solução geral reutilizável para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software.

Nenhum padrão de projeto é um esqueleto pronto e definitivo de solução para um problema do dia a dia, ele é um modelo. Não os use como verdade absoluta, a única verdade absoluta é que não existem verdades absolutas.

O que é GoF ou Gang of Four (Gangue dos quatro)

A GoF é formada por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, que juntos escreveram um livro chamado Design Patterns: Elements of Reusable Object-Oriented Software em 1994. O livro ganhou muita popularidade e inspirou muitos outros depois.

Eles estabeleceram 23 padrões de projeto (eu vou passar disso aqui, espero) separados em 3 categorias, padrões de criação, estruturais e comportamentais.

** Vou colocar um asterisco nos padrões definidos originalmente pela GoF.**

A medida que os artigos sobre cada padrão for escrito vou colocando os links.

Padrões de criação

Focados na criação de objetos

- Abstract Factory*
- Builder*
- Factory Method*
- Multiton (Considerado anti-pattern!)
- Pool
- Prototype*
- SimpleFactory
- Singleton* (Considerado an anti-pattern!)

- StaticFactory

Padrões estruturais

Focados na associação entre objetos

- Adapter*
- Bridge*
- Composite*
- DataMapper
- Decorator*
- DependencyInjection
- Facade*
- FluentInterface
- Flyweight*
- Proxy*
- Registry

Padrões comportamentais

Focados nas interações entre objetos

- Chain Of Responsibilities*
- Command*
- Interpreter*
- Iterator*
- Mediator*
- Memento*
- NullObject
- Observer*
- Specification
- State*
- Strategy*
- TemplateMethod*
- Visitor*

Mais

- Delegation
- ServiceLocator
- Repository
- EAV

Conclusão

Claro que ainda existem outros, mas a lista já ficou grande e isso me da muito o que escrever, então vou parar por aqui, quem sabe no futuro eu coloque mais sobre o assunto aqui.

https://webdevbr.com.br/padroes-de-projeto-no-php

Introdução aos Padrões de Projeto com PHP

por Rafael Jaques

Atenção! Essa postagem foi escrita há mais de 2 anos. Na informática tudo evolui muito rápido e algumas informações podem estar desatualizadas. Embora o conteúdo possa continuar relevante, lembre-se de levar em conta a data de publicação enquanto estiver lendo. Caso tenha sugestões para atualizá-la, não deixe de comentar!

Salve, galera! Depois de quase 5 meses sem dar as caras por aqui, consegui tempo pra voltar! Realmente o final de ano foi difícil, mas graças a Deus, terminei a faculdade e agora tenho mais tempo para me dedicar aos estudos que antes eram paralelos.

Mas sem muita conversa, vamos direto ao que interessa: padrões de projeto.

Hoje inicio uma série de alguns posts falando sobre os padrões de projeto mais úteis para programadores PHP e algumas aplicações práticas dos mesmos. Então vamos estudar juntos!

Antes de começar…

Para iniciar os seus estudos, antes de mais nada, você deve estar familiarizado com o PHP 5 (ou superior). Assumo que você já tenha fluência na linguagem. Também é necessário que você já tenha uma boa noção de Orientação a Objetos. Todos os padrões de projeto são feitos de forma orientada a objetos e os termos abordados são diretamente ligados a este paradigma. Caso você não conheça esta metodologia ou nunca tenha trabalhado com ela, sugiro alguns links para dar uma olhada:

- Programação Orientada a Objetos: uma introdução
- Programação Orientada ao Objeto: uma abordagem didática
- Tutorial de Programação Orientada a Objetos no PHP
- PHP Orientado a Objetos: Para quem está começando

O que são e para que servem os Padrões de Projeto?

O conceito de padrões de projeto foi originalmente concebido pelo arquiteto austríaco Christopher Alexander, em meados da década de 1970. O que levou Alexander a realizar este estudo foi o fato de que ele gostaria de descobrir se havia alguma regra que pudesse especificar quando uma construção é de boa ou de má qualidade.

A partir deste estudo, Alexander começou a levantar pontos que estavam presentes em construções boas e não estavam nas ruins e vice versa. Com estes dados em mãos ele começou a elaborar diversas propostas para a solução de problemas comuns dentro da engenharia e da arquitetura.

> “Cada padrão descreve um problema que ocorre frequentemente em nosso ambiente e então descreve a essência da solução deste problema, de tal forma que você possa usar a mesma solução milhões de vezes, sem nunca utilizá-la da mesma forma.”
>
> Christopher Alexander – A Pattern Language

Tendo isso em mente podemos afirmar que um padrão de projeto nada mais é do que uma forma prevista, estruturada e documentada de solucionar um dado problema que é bastante comum.

No início da década de 1990, a GoF (Gang of Four – Gangue dos Quatro), formada por Eric Gamma, Richard Helm, Ralph Johnson e John Vlissides percebeu que as técnicas publicas por Alexander iam muito além do objetivo de padronizar a criação arquitetônica. Era possível identificar padrões que iriam auxiliar na resolução dos problemas comumente encontrados durante o desenvolvimento de softwares. Lançaram então o **Design Patterns: Elements of Reusable Object-Oriented Software**, livro este que tornou-se a maior influência sobre a comunidade de desenvolvimento de softwares da época (e que perdura até hoje). Ao todo são citados 23 padrões, divididos em três categorias: criacionais, estruturais e comportamentais. Estas categorias serão explicadas mais adiante.

Por que estudar Padrões de Projeto?

Agora que você já sabe o que são os padrões de projeto, já deve estar procurando (se ainda não encontrou) motivos para estudá-los. Eis alguns bons motivos de porque você deve investir tempo de estudo nestes padrões:

- Reutiliza ao invés de redescobrir: todos os padrões já foram previamente testados e implementados em projetos que deram certo.
- Otimiza a comunicação entre os desenvolvedores: unifica a língua que é falada entre os envolvidos no projeto, pois todos tem ciência dos termos empregados.
- Facilita a análise: Quando você iniciar a análise já terá base e estará bem direcionado ao que implementar no seu projeto;
- Proporciona alterações menos dolorosas: uma vez que existe uma padronização, as alterações tornam-se mais simples devido às estruturações do projeto;
- Desestimula o uso de herança: delegar é melhor do que estender – a GoF não condena de maneira alguma o uso de herança, mas recomenda que se dê preferência pela extensão por delegação caso queira escrever código reusável. A herança não é totalmente desencorajada, até porque um bom sistema usa tanto delegação quanto herança;
- Aumenta a confiabilidade;
- Reduz a complexidade do código;

Como dividem-se os padrões?

Para uma melhor organização dos padrões apresentados, eles foram divididos conforme a aplicação de cada um. As três categorias distintas dos padrões de projeto são:

- **Criacionais:** Abstraem e escondem o processo de instanciação. O sistema torna-se independente da maneira como o objeto é composto, construido e representado.
- **Estruturais:** Trabalham com a composição de classes e objetos, definindo a relação entre ambos.
- **Comportamentais:** Definem como os objetos irão se comportar e padroniza a comunicação que haverá entre os objetos dentro da estrutura do projeto.

E agora?

E agora, meus caros amigos, teremos que esperar pelo próximo post, onde vou começar a falar mais sobre os tipos de padrões, explicar melhor as divisões e dar um resumo de cada um.

Espero que esse post tenha servido para dar uma iluminada no conceito de Design Patterns que muitas vezes é bastante obscuro!

Aceito sugestões para as próximas postagens!

Um abraço a todos e fiquem com Deus!
Rafael Jaques

http://www.phpit.com.br/artigos/introducao-aos-padroes-de-projeto-com-php.phpit

Introdução aos Padrões de Projeto com PHP – Parte 2

por Rafael Jaques

Atenção! Essa postagem foi escrita há mais de 2 anos. Na informática tudo evolui muito rápido e algumas informações podem estar desatualizadas. Embora o conteúdo possa continuar relevante, lembre-se de levar em conta a data de publicação enquanto estiver lendo. Caso tenha sugestões para atualizá-la, não deixe de comentar!

E aí, povo! Tudo certo?

Continuando a nossa série de estudos sobre Padrões de Projeto, hoje daremos uma olhada mais aprofundada nos padrões que existem e que iremos estudar. Caso você ainda não tenha lido, dê uma olhada na primeira parte dessa série dando uma introdução aos design patterns.

Uma vez que os padrões de projeto foram desenvolvidos originalmente para suprir necessidades de software independente da linguagem, alguns autores gostam de selecionar apenas os padrões mais utilizados em determinada linguagem. Nesta série também não citarei todos os padrões da GoF porque alguns, como o Iterator, já estão implementados nativamente na linguagem (o Iterator é um foreach, por exemplo). Aproveitarei o espaço aberto por esses padrões e indicarei alguns outros que não fazem parte da GoF mas que são muito importantes de serem estudados.

Vamos dar uma olhada em como se dividem os padrões da GoF e em seguida quais serão alguns dos padrões citados mais adiante.

Padrões da Gang of Four

Criacionais

- **Abstract Factory:** Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
- **Builder:** Separa a construção de um objeto da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

- **Factory Method:** Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada. O Factory Method permite a uma classe postergar a instanciação às subclasses.
- **Prototype:** Especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando este protótipo.
- **Singleton:** Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.

Estruturais

- **Adapter:** Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis.
- **Bridge:** Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.
- **Composite:** Compõe objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.
- **Decorator:** Atribui responsabilidades adicionais a um objeto dinamicamente. Os decorators fornecem uma alternativa flexível a subclasses para extensão da funcionalidade.
- **Façade:** Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O Façade define uma interface de nível mais alto que torna o subsistema mais fácil de usar.
- **Flyweight:** Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.
- **Proxy:** Fornece um objeto representante (surrogate), ou um marcador de outro objeto, para controlar o acesso ao mesmo.

Comportamentais

- **Chain of Responsability:** Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
- **Command:** Encapsula uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre (log) solicitações e suporte operações que podem ser desfeitas.
- **Interpreter:** Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem.
- **Iterator:** Cria um iterador para interagir com os objetos, retirando a iteração das responsabilidades do próprio objeto.
- **Mediator:** Define um objeto que encapsula como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.
- **Memento:** Sem violar a encapsulação, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado.
- **Observer:** Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.

- **State:** Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.
- **Strategy:** Define uma família de algoritmos, encapsula cada um deles e os faz intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.
- **Template Method:** Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
- **Visitor:** Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O Visitor permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera, criando métodos virtualmente.

Além das divisões por finalidade, os padrões de projeto possuem outro tipo de divisão. Existem padrões de classe e padrões de objeto.

Outros padrões de projeto

O fato de estes padrões não terem sido desenvolvidos pela GoF não os torna menos importantes. Inclusive muitos destes padrões você já pode ter implementado, de uma forma ou de outra, e nem ter percebido!

Os padrões que veremos a seguir são mais específicos do que os vistos anteriormente. Suas finalidades são focadas na resolução de problemas não tão genéricos e na aceleração do desenvolvimento.

Vamos dar uma olhada:

- **Table Data Gateway:** Cria objetos que referenciam tabelas no banco de dados dando acesso dando acesso simplificado à elas sem a necessidade de digitar SQL.
- **Row Data Gateway:** Dá acesso a um determinado registro em uma tabela do banco e permite que sejam alterados seus valores.
- **MVC:** O nome deste padrão é Model-view-controller. Separa a aplicação em três camadas: banco de dados, lógica de negócio e apresentação.
- **Active Record:** Deriva-se do Row Data Gateway. Mapeia uma linha de registro de uma determinada tabela no banco de dados e disponibiliza métodos de acesso, gravação e alteração para o mesmo, implementando algumas regras de negócio.
- **Data Mapper:** Uma versão aprimorada do Active Record, porém com mais camadas de separação para lidar com lógicas de negócio mais complexas.
- **Lazy Load:** Permite que objetos encadeados só sejam carregados quando necessário, reduzindo a carga na memória.

Conclusão

E por hoje é só, pessoal! Aproveitei esse artigo pra falar um pouquinho mais sobre alguns padrões que vamos estudar daqui pra frente e espero que tenha servido para dar uma iluminada nas ideias! Provavelmente no próximo artigo já começaremos a colocar a mão na massa! :-)

Um abraço para todos e fiquem com Deus!