Coordenadoria de Tecnologia da Informação

Tecnologia em Análise e Desenvolvimento de Sistemas

Lamberjack’s ORM

*Diego Francklin Martins dos Santos*

Sorocaba

Junho – 2018

Coordenadoria de Tecnologia da Informação

Tecnologia em Análise e Desenvolvimento de Sistemas

Lamberjack’s ORM

Diego Francklin Martins dos Santos

Prof. Cristiane Palomar Mercado - Orientadora

Sorocaba

Junho – 2018

**Dedicatória**

Dedico este trabalho aos meus pais, amigos e esposa que sempre me ajudaram e incentivaram a seguir adiante e vencer todas as barreiras e desafios.

**Agradecimento**

Agradeço a todos que de uma forma ou de outra colaboraram para com este trabalho.

**Sumário**

[Resumo 1](#_Toc516420606)

[1. Objetivo 2](#_Toc516420607)

[2. Introdução 4](#_Toc516420608)

[3. *Framework* 5](#_Toc516420609)

[4. *Design Patterns* 7](#_Toc516420610)

[4.1. Quando não usar? 8](#_Toc516420611)

[4.2. Qual a diferença entre o *Design Pattern* e o *Framework* 8](#_Toc516420612)

[5. ORM 9](#_Toc516420613)

[5.1. *Active Record* 9](#_Toc516420614)

[5.2. *Data Mapper* 10](#_Toc516420615)

[6. Estudo de caso 11](#_Toc516420616)

[6.1. Tecnologias 11](#_Toc516420617)

[6.2. Mapeamento 11](#_Toc516420618)

[6.3. Conexões 16](#_Toc516420619)

[6.4. *Drivers* 18](#_Toc516420620)

[6.4.1. Criar Conexões 19](#_Toc516420621)

[6.4.2. Tipos de Dados 19](#_Toc516420622)

[6.4.3. Valores Auto Gerados 20](#_Toc516420623)

[6.4.4. Paginação 20](#_Toc516420624)

[6.5. Criação das Tabelas 21](#_Toc516420625)

[7. Codificação 24](#_Toc516420626)

[8. Considerações Finais 25](#_Toc516420627)

[Referências 26](#_Toc516420628)

[Glossário 27](#_Toc516420629)

**Lista de Figuras**

[Figura 1: Estrutura do *Framework* 2](#_Toc516420630)

**Lista de Listagens**

[Listagem 1 – Classe Pessoa 12](#_Toc516420631)

[Listagem 2 – Exemplo de uso da classe ReflectionClass 13](#_Toc516420632)

[Listagem 3 – Exemplo da classe ReflectionProperty 13](#_Toc516420633)

[Listagem 4 – Exemplo de Expressão Regular 14](#_Toc516420634)

[Listagem 5 – Exemplo de arquivo de conexões 17](#_Toc516420635)

[Listagem 6 – Exemplo para utilizar a conexão 18](#_Toc516420636)

[Listagem 7 – Exemplo para configurar a criação de tabelas 21](#_Toc516420637)

[Listagem 8 – Exemplo para configurar para apagar as tabelas 22](#_Toc516420638)

[Listagem 9 – Exemplo para executar ações antes de apagar as tabelas e depois de cria-las 23](#_Toc516420639)

**Lamberjack’s ORM**

# Resumo

Durante a criação e manutenção de projetos é investido um grande esforço e uma grande quantidade de tempo para analisar, criar e integrar bancos de dados à aplicação. E geralmente, torna-se muito trabalhoso migrar para outros bancos de dados e com grande potencial de se tornar um verdadeiro pesadelo.

O ORM (*Object-Relational Mapping*) é uma técnica que visa o mapeamento entre o mundo orientado à objetos (os nossos modelos) e o mundo relacional (o banco de dados).

Essa técnica está em crescente uso no mercado pois permite abstrair a utilização de comandos SQL, não cria dependência com uma marca de banco de dados específica e pode-se observar um ganho de produtividade nas tarefas diárias.

Diversos *frameworks* surgiram no mercado nos últimos anos escritos em diversas linguagens de programação para facilitar a utilização dessa técnica. Porém alguns possuem certas limitações que este trabalho pretende remover.

# Objetivo

O objetivo deste trabalho é desenvolver um *framework* chamado Lumberjack’s ORM para facilitar a utilização de bancos de dados relacionais em aplicações desenvolvidas em PHP, permitindo que estas eliminem por completo a dependência com a linguagem SQL e com um banco de dados de marca específica, tornando simples a configuração e integração com os bancos de dados mais comuns do mercado, de forma que o *framework* forneça um suporte à diversas funcionalidades que visam simplificar tarefas, que antes seriam feitas manualmente e ocupariam muito tempo; tempo este que poderia ter um aproveitamento mais significativo se focado em tarefas mais importantes do projeto.

Figura 1: Estrutura do *Framework*



Fonte: Autor, 2018

O *framework* conta com as seguintes funcionalidades em sua arquitetura:

* Mapeamento de entidades (tabelas ou *views*) utilizando *Annotations*, possibilitando também que seja mapeado todos os tipos de relacionamentos entre as entidades;
* Armazenamento e utilização de múltiplas conexões com bancos de dados;
* Possibilitar a criação de tabelas no banco de dados caso não existam baseando-se no mapeamento dos modelos;
* Persistência de dados de novos registros ou registros existentes;
* Deleção de dados existentes no banco;
* Criação de consultas possibilitando a utilização de quaisquer tipos de relacionamentos, filtros, ordenações, agrupamentos e funções de agregação.

Para fins de demonstração das funcionalidades do *framework*, será criada uma aplicação simples.

# Introdução

Para desenvolver um software de qualidade, é preciso pensar em manutenção e evolução. Desenvolvedores lidam com diversas variáveis e resolução de diversos problemas ao longo de um dia de trabalho. Muitos desses problemas que surgem são problemas repetidos, e ao invés de desenvolver novas soluções, eles reutilizam soluções que funcionaram no passado e as utilizam repetidamente em seus projetos.

Por isso a importância dos *design patterns (*padrões de projetos), que proporcionam reaproveitamento de soluções para projetos e não apenas a reutilização de código.

Atualmente no mercado existem alguns *frameworks* ORM em PHP, porém, vários tem problemas com documentação ou é insuficiente no que diz respeito a relacionamentos entre tabelas, principalmente em um relacionamento N para N.

Durante a criação de novos projetos, exige do desenvolvedor uma preocupação extra no que se relaciona a banco de dados, pois, sem a utilização de um *framework* ORM as aplicações ficam com muita dependência de um determinado banco em si, o que torna a manutenção e/ou evolução da aplicação muito mais complicada de ser realizada, por exemplo, para migrar para um outro banco de dados mais robusto no futuro. Sendo assim, como estudo de caso, será apresentada as funcionalidades do *framework* para a persistência e manipulação de suas informações

Ao final do projeto, o ORM desenvolvido será disponibilizado gratuitamente à comunidade de desenvolvedores através de um repositório público no GitHub, uma vez que possua uma versão estável.

# *Framework*

Com os avanços tecnológicos na área de desenvolvimento de software, vem surgindo diversas abordagens para reduzir a complexidade do processo de produção de um software e aumentar a qualidade do produto e a produtividade da equipe de desenvolvimento, e uma dessas abordagens é a reutilização de código ou soluções, com o objetivo de evitar a resolução de cada problema partindo do básico ou mesmo do zero.

Segundo Gamma (1995), uma coisa que os projetistas de software mais experientes sabem é que não devem resolver cada problema a partir de princípios elementares ou do zero, mas reutilizar soluções que funcionaram no passado.

Essa ideia de reutilização não é nova nem exclusiva para o desenvolvimento de software, ela surgiu quando começaram a encontrar soluções consistentes e que poderiam ser aplicadas a novos problemas. O uso frequente dessas soluções faz com que fiquem conhecidas e se tornem aceitas, generalizadas e padronizadas.

De acordo com os princípios da Engenharia de Software, principalmente em termos de reutilização, a Orientação a Objetos surgiu como poderoso instrumento para o desenvolvimento de software. De acordo com Jones (2001), os objetos tornaram-se os blocos de construção onipresentes do software moderno e a orientação a objetos o paradigma dominante da era contemporânea.

O *framework* é uma técnica da Orientação a Objetos que é voltada exclusivamente para o reaproveitamento de soluções que se beneficia de três das características que linguagens de programação orientadas à objetos oferecem: herança, polimorfismo e abstração.

Um *framework* engloba uma funcionalidade específica e comum a várias aplicações. Ele define a organização de um software, os tipos de objetos e as interações entre esses objetos. Ele pode ser considerado o esqueleto no qual o sistema se sustenta.

Os *frameworks* possuem interfaces complexas e são de fácil customização. Em sua composição contam com interfaces e classes abstratas e seu uso se dá através da especialização ou composição dos seus serviços.

Por isso, pode-se afirmar que os *frameworks* possuem pontos fixos, que são os serviços e componentes implementados nele, os quais realizam chamadas indiretas aos pontos extensíveis, que são serviços e funcionalidades que devem ser implementados por meio de herança, onde os desenvolvedores irão inserir os códigos relacionados ao domínio do problema da aplicação.

Segundo Fayad (1999), a utilização de *frameworks* apresenta os seguintes benefícios:

I. Melhora a modularização – encapsulamento dos detalhes voláteis de implementação através de interfaces estáveis.

II. Aumenta a reutilização – definição de componentes genéricos que podem ser replicados para criar novos sistemas.

III. Extensibilidade – favorecida pelo uso de métodos “gancho” que permitem que as aplicações estendam interfaces estáveis.

IV. Inversão de controle –o código do desenvolvedor é chamado pelo código do framework. Dessa forma, o framework controla a estrutura e o fluxo de execução dos programas.

# *Design Patterns*

Segundo Gerra (2012), um *design pattern* (padrão de projeto) é composto por três partes: um contexto, um problema e uma solução. O contexto define o ambiente e as circunstâncias dentro do qual algo existe. O problema é uma questão indefinida, algo que precisa de investigação e solução, e que está diretamente relacionado ao contexto. E por fim a solução, diz respeito a resposta do problema que ajuda a soluciona-lo.

Um *design pattern* descreve uma solução para um problema em um contexto, porém não descreve qualquer solução, mas sim, uma solução que já tenha sido utilizada com sucesso diversas vezes em vários contextos diferentes. Por isso, podemos dizer que um padrão não descreve soluções novas, mas soluções já testadas e consolidadas. Mas para ser um padrão, uma solução não basta ser recorrente, mas precisar ser uma boa solução.

Segundo Christopher Alexander (1977), “cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca o fazer da mesma maneira”.

Os padrões popularizaram-se com o conhecido livro “*Design Patterns: Elements of Reusable Object-Oriented Software*”. O livro é um catálogo que possui 23 *design patterns*, cada uma descrevendo uma solução para problemas de software orientado a objeto, o contexto e suas eventuais consequências. As soluções são utilizadas até hoje por desenvolvedores em todo o mundo.

A primeira definição do que seria um design pattern foi apresentada no livro “*A Times Way of Building*” (Oxford University Press, 1979), que é: “Cada padrão é uma regra de três partes, que expressa uma relação entre um certo contexto, um problema e uma solução”.

## Quando não usar?

Quanto mais *design patterns* eu utilizar, melhor meu projeto vai ficar? Não! Um padrão é uma solução para um problema, portanto, utilizá-lo onde não há um problema apenas complica mais as onde não há necessidade.

Um *design pattern* também possui consequências negativas que podem se sobrepor às vantagens em alguns casos. Sua utilização desnecessária pode ser desastrosa e isso, no mínimo, aumenta a complexidade do sistema sem necessidade, o que torna muito mais difícil a manutenção do mesmo. Esse uso inadequado ou o uso exagerado é o que constitui o chamado *anti-pattern* (antipadrão). De acordo com Andrew Koenig (2018), se um padrão representa a “melhor prática”, então um *anti-pattern* representa uma “lição aprendida”.

## Qual a diferença entre o *Design Pattern* e o *Framework*

O conceito de design pattern é muito parecido ao do *framework*, e isso gera frequentemente confusão sobre o que caracteriza cada um e suas particularidades.

Para Fowler, (1997), os padrões descrevem maneiras comuns de fazer as coisas e são coletados por pessoas que identificam temas repetitivos em projetos. Essas pessoas identificam cada tema e o descrevem de modo que outras pessoas possam ler o padrão e ver como aplicá-los.

Padrões documentam *frameworks* e ajudam a garantir o uso correto de sua funcionalidade. Para Johnson (1997), padrões são elementos microarquiteturas de *frameworks*. Um *framework* usualmente utiliza muitos padrões, ou seja, padrões são menores do que muito *frameworks*, podendo ser vistos como seus blocos construtores, ou seja, as partes constituintes de um *framework*.

# ORM

Para trabalhando com dados em uma aplicação, você provavelmente precisará de um ORM. Nesse capítulo será abordado o que é um ORM e quais os tipos mais comuns.

ORM (da sigla em inglês que significa *Object-Relational Mapping*, traduzindo para português Mapeamento Objeto-Relacional) é a camada de aplicação que está situada entre a camada de acesso a dados da aplicação e o banco de dados. Ele é responsável por tomar conta de boa parte do trabalho pesado criando, atualizando, deletando e consultando dados do bando de dados.

Em uma linguagem de programação orientada a objetos, o principal ponto de referência para os desenvolvedores é o objeto. No entanto, os bancos de dados trabalham com a estrutura de tabelas e registros, em sua forma mais simples (textos e números).

Trabalhando com objetos, uma única estrutura para armazenar as informações e relacionamentos nas propriedades do mesmo objeto. No bando de dados, as informações são armazenadas em registros através de uma ou várias tabelas.

Pode-se dizer então que o ORM é a camada “mágica” que transforma os dados armazenadas na forma de objetos para a estrutura das tabelas utilizadas pelo banco de dados e vice-versa.

Normalmente não é necessário se preocupar com o tipo de ORM que está sendo utilizado para desenvolver a aplicação, porém, conforme o desenvolvedor se aprofunda em como aplicações são desenhadas, vale a pena explorar um pouco mais a fundo os tipos de ORM disponíveis para uso.

Os tipos mais comuns são o padrão *Active Record* e o padrão *Data Mapper*.

## *Active Record*

De acordo com Fowler (2018), dentro do padrão *Active Record*, cada objeto do modelo precisa saber como se comunicar com o banco de dados e como persistir suas informações, ou seja, o objeto do modelo contém a lógica para abrir uma conexão com o banco de dados, criar, alterar e deletar dados do banco de dados. Portanto, não existe uma camada específica para que essas tarefas sejam executadas.

Em sistemas mais simples, o desenvolvimento se torna mais rápido e leve, porém, existe uma dependência muito grande entre o modelo e o esquema do banco de dados. Isso faz com que a manutenção seja mais complicada, por algumas razões: qualquer mudança em um, implica diretamente na mudança do outro; quanto mais complexos os modelos são, mais complexo se torna o mapeamento; aumenta muito a duplicação de código pois cada modelo acaba possuindo logicas de como persistir os dados.

## *Data Mapper*

Segundo Fowler (2018), o padrão *Data Mapper* consiste em uma camada de objetos que mapeiam, ou seja, movem dados entre os objetos do modelo e o banco de dados, mantendo um independente do outro. O uso desse padrão torna o código mais limpo e simples de se compreender.

Uma implementação simples de *Data Mapper*, mapeia uma tabela do banco de dados para uma classe equivalente, campo a campo.

Os objetos do modelo geralmente estão muito interconectados, por isso é necessário utilizar um padrão que limita até qual nível desses relacionamentos é desejado alcançar, do contrário, seria possível trazer muito mais dados do banco de dados do que o desejado. Esse padrão é denominado *Lazy Load*.

Fowler (2018) define *Lazy Load* como um objeto que não contém todos os dados que você precisa, ou seja, o objeto não está completo, mas sabe como obter os dados.

Para o desenvolvimento desse trabalho, esse foi o padrão escolhido.

# Estudo de caso

O foco desse trabalho é desenvolver um *framework* que atenda às necessidades dos programadores, com relação ao mapeamento de relacionamento entre tabelas dos bancos de dados e concilie com a facilidade de utilização.

## Tecnologias

Para o desenvolvimento do ORM a linguagem PHP foi escolhida. PHP é uma sigla em inglês que significa *Hypertext PreProcessor* (ou em português Pré-Processador de Hipertexto).

O PHP é uma linguagem de *script*, largamente utilizada, principalmente para desenvolvimento web. Seu uso é extremamente simples para iniciantes e oferece diversos recursos para desenvolvedores profissionais.

Outro motivo para o grande uso do PHP é o fato de ser *open source* e poder ser utilizado na maioria dos sistemas operacionais, incluindo Linux, variantes Unix, Mac OS e Microsoft Windows.

## Mapeamento

Para realizar o mapeamento dos modelos de dados, foi utilizado o conceito de *Annotations*.

As *annotations* são “etiquetas” que adicionam metadados relevantes sobre classes, métodos e propriedades. Ou seja, através do uso de *annotations*, pode-se adicionar às classes informações para mapear tabelas do banco de dados e adicionar às propriedades da classe para mapear as colunas de uma tabela do banco de dados, para que posteriormente, em tempo de execução, os metadados indicados pelas *annotations* sejam analisados e a partir disso, o ORM irá trabalhar de acordo com essas informações.

Porém, no PHP, como em outras linguagens de programação como o Java, não existe uma estrutura nativa para adicionar esses metadados dentro do contexto da classe. Então, qual seria a solução para isso? Utilizar expressões regulares e a família de classes *Reflection* do PHP para resolver essa questão.

As linguagens de programação possuem uma estrutura para documentar as classes, propriedades e métodos, essa estrutura é o bloco de comentário. No PHP, essa estrutura é comumente conhecida como *PHPDoc*.

Para obter os comentários contidos nas classes e extrair as informações dos comentários, pode-se utilizar a família de classes *Reflection*, no caso, especificamente as classes *ReflectionClass* e *ReflectionProperty*.

Para os exemplos a seguir, considere a seguinte classe do modelo de dados:

1. <?php
2. namespace App;
3. /\*\*
4. \* @ORM/Entity
5. \* @ORM/Table(name=pessoa)
6. \*/
7. class Pessoa {
8. /\*\*
9. \* @ORM/Id
10. \* @ORM/Generated
11. \* @ORM/Column(name=pessoa\_id, type=int)
12. \*/
13. public $id;
14. /\*\*
15. \* @ORM/Column(name=nome, type=string, length=50)
16. \*/
17. public $nome;
18. /\*\*
19. \* @ORM/Column(name=data\_nasc, type=date)
20. \*/
21. public $dataNasc;
22. }

Listagem 1 – Classe Pessoa

Fonte: Autor, 2018

Extrair o *PHPDoc* de uma classe do modelo é realizado de maneira simples através do uso da classe *ReflectionClass*. Como no exemplo abaixo:

1. <?php
2. $reflection = new ReflectionClass(‘App\Pessoa’);
3. $doc = $reflection->getDocComment();
4. ?>

Listagem 2 – Exemplo de uso da classe ReflectionClass

Fonte: Autor, 2018

No exemplo mostrado na Listagem 2, na linha 2, é criada a instância da classe *ReflectionClass*, passando como parâmetro para o construtor, o nome da classe do modelo. Já na linha 3, o método *getDocComment* retorna o bloco de comentário da classe informada no construtor.

Com a instância da classe *ReflectionClass*, é possível acessar as propriedades da classe do modelo. Por exemplo:

1. <?php
2. $reflection = new ReflectionClass(‘App\Pessoa’);
3. $doc = $reflection->getDocComment();
4. $properties = $reflection->getProperties();
5. foreach($properties as $property) {
6. $propertyDoc = $property->getDocComment();
7. }
8. ?>

Listagem 3 – Exemplo da classe ReflectionProperty

Fonte: Autor, 2018

Na linha 4, o método *getProperties* retorna um *array* contento instancias da classe *ReflectionProperity*, uma para cada propriedade contendo da classe. Então, para obter os blocos de comentário das propriedades da classe, percorre-se a o *array* na linha 6 e na linha 7 o método *getDocComment* retorna o bloco de comentário da propriedade atual.

Agora, com os blocos de comentário em mãos, o próximo passo é utilizar a expressões regulares para extrair as *annotations* e suas informações.

Expressão Regular é um recurso muito comum em diversas linguagens. Para executar uma expressão regular no PHP, usa-se duas das diversas funções nativas disponíveis chamadas “preg\_match” e “preg\_match\_all” (PHP, 2018). Ambas executam uma expressão regular contra um texto procurando por referências. A diferença entre as duas é que a função “preg\_match” pára quando encontra a primeira correspondência e a função “preg\_match\_all” procura no texto inteiro e retorna todas as correspondências.

Na Listagem 1, a classe “Pessoa” possui algumas *annotations*, entre elas a *annotation* “@ORM/Table”, que informa, entre outras coisas, o nome da tabela em questão que deve ser mapeada. Então para identificar essa *annotation* e suas propriedades, para então extrair suas informações, faz-se da seguinte maneira:

1. <?php
2. $reflection = new ReflectionClass(‘App\Pessoa’);
3. $doc = $reflection->getDocComment();
4. $findAnnotatoins = ‘/@ORM\/[@A-Za-z0-9=,\_\/\s\(\)\{\}]+/i’;
5. $findTable = ‘/Table\([@A-Za-z0-9=,\_\/\s\(\)]+\)/i’;
6. $findName = ‘/name[\s]?=[\s]?(\w+)/i’;
8. preg\_match\_all($findAnnotatoins, $doc, $annotations);
9. $annotations = join($annotations[0], ‘’);
10. preg\_match($findTable, $annotations, $table);
11. preg\_match($findName, $table[0], $name);
12. ?>

Listagem 4 – Exemplo de Expressão Regular

Fonte: Autor, 2018

As linhas 2 e 3 já foram mostradas anteriormente, mas nas linhas 4, 5 e 6, estão declaradas as expressões regulares que serão usadas para identificar a *annotation* e extrair as informações dela. Para um melhor entendimento das expressões regulares, ela pode ser quebrada em algumas partes menores:

* **“/”**

A primeira barra indica o começo da expressão regular e a última indica o começo da expressão regular;

* **“@ORM\/”**

Esse trecho indica que o texto correspondente precisa, obrigatoriamente, começar com o texto “@ORM/”;

* **“[]”**

O trecho “[]” indica um conjunto, ou seja, a expressão deve considerar qualquer combinação dos caracteres entre colchetes como um texto válido;

* **“@A-Za-z0-9=,\_\/\(\)\{\}”**

O trecho acima indica que a expressão regular deve considerar o caractere “@”, todas letras maiúsculas e minúsculas, os números de 0 a 9, os caracteres “=”, “,”, “/”, “(“, “)”, “{“, “}”;

* **“\s”**

O trecho “\s” indica que a expressão regular deve considerar os caracteres de espaço e tabulação;

* **“\w”**

O “\w” representa uma palavra com letras, números e underline;

* **“i”**

A letra “i” sinaliza que a expressão regular deve ser *case insensitive*, ou seja, a expressão regular não deve diferenciar entre letras maiúsculas e minúsculas;

* **“+”**

O “+” indica que a expressão deve que o padrão que o precede ocorra uma ou mais vezes, por exemplo, “\w+”, o padrão indica que a expressão deve esperar uma ou mais palavras dentro do texto em que a expressão será aplicada;

* **“?”**

O “?” indica que a expressão deve que o padrão que o precede ocorra zero ou uma vez, por exemplo, “\w?”, o padrão indica que a expressão deve esperar zero ou uma (no caso a primeira) palavra dentro do texto em que a expressão será aplicada.

A combinação dessas partes e de várias outras possíveis é o que cria o padrão a ser encontrado dentro do texto. As funções “preg\_match” e “preg\_match\_all” utilizam a expressão, que é informada no primeiro parâmetro da função, para encontrar as ocorrências dentro do bloco de texto, informado no segundo parâmetro e armazena as ocorrências em um *array* que é armazenado na variável informada no terceiro parâmetro.

Com todas as informações identificadas, elas serão armazenadas em um *array* acessível pelo ORM, para que posteriormente, sejam usadas para realizar os mapeamentos de consultas simples, consultas com relacionamentos (para identificar as colunas usadas para relacionar as duas tabelas), inclusões, alterações e exclusões na tabela a ser mapeada pela classe do modelo.

A leitura e extração das informações de classes para o mapeamento é realizado sob demanda, isso quer dizer que, o mapeamento de uma entidade irá ocorrer somente no momento em que o ORM precise trabalhar com a classe do modelo.

Por exemplo, considerando a classe “Pessoa” na Listagem 1, para realizar uma consulta à tabela “pessoa”, o ORM lerá o mapeamento dessa classe somente no momento em que precisar montar a consulta pela primeira vez. Uma vez que o mapeamento esteja completo, essas informações estão disponíveis até o final do processamento do *script* PHP.

## Conexões

Para que se possa enviar comandos ao banco de dados e receber as respostas desses comandos, é necessário que exista uma conexão com o banco de dados para ser possível realizar essa comunicação.

As conexões que serão utilizadas pelo ORM devem ser declaradas em um arquivo com extensão “.php”. Por padrão, o arquivo é esperado que esteja na pasta raiz do ORM com o nome “connection.config.php”, ou seja, supondo que o ORM esteja localizado “/home/user/app/orm/”, então o caminho para o arquivo seria “/home/user/app/orm/connection.config.php”.

O arquivo deve conter um *array* com uma ou mais conexões, onde a chave da conexão é o nome identificador da conexão e o valor é um *array* contendo as informações da conexão. As informações variam de acordo com o banco de dados a ser utilizado.

Na Listagem 5, pode-se ver um exemplo de arquivo de conexões para melhor compreensão.

1. <?php
2. return [
3. ‘exemplo-mysql’ => [
4. ‘db’ => ‘mysql’,
5. ‘version’ => ‘5.7.11’,
6. ‘host’ => ‘localhost’,
7. ‘schema’ => ‘app’,
8. ‘user’ => ‘root’
9. ‘pass’ => ‘root’
10. ],
11. ‘exemplo-sqlite’ => [
12. ‘db’ => ‘sqlite’,
13. ‘version’ => ‘3’,
14. ‘file’ => ‘../data/app-storage.sq3’,
15. ],
16. ];
17. ?>

Listagem 5 – Exemplo de arquivo de conexões

Fonte: Autor, 2018

De acordo com o exemplo da Listagem 5, duas conexões são definidas, na linha 3 está a declaração de uma conexão com o nome “exemplo-mysql” e na linha 11 a conexão com o nome “exemplo-sqlite”.

Os valores para a conexão “exemplo-mysql” é um *array* contendo as chaves “db”, “version”, “host”, “schema”, “user” e “pass”. A chave “db” contém o banco de dados a ser utilizado. A chave “version” indica a versão do banco de dados utilizado, no qual o *Driver* deve corresponder à essa versão. A chave “host” é o endereço onde o banco de dados está localizado. A chave “schema” é o banco de dados (conjunto de tabelas) que será utilizado. As chaves “user” e “pass” são respectivamente o usuário e a senha de acesso ao banco de dados.

Os valores para a conexão “exemplo-sqlite” é um *array* contendo as chaves “db”, “version”, “file”. As chaves “db” e “version” funcionam da mesma maneira que a conexão anterior. A chave “file” indica o arquivo local o qual o banco de dados SQLite utilizará para armazenar os dados.

Para informar ao ORM qual (ou quais) conexão será utilizada na aplicação, faz-se através da classe principal do ORM conforme o exemplo a seguir:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’);
4. ?>

Listagem 6 – Exemplo para utilizar a conexão

Fonte: Autor, 2018

A classe “ORM\Orm” concentra entre outras informações, as conexões com o banco de dados, e para isso ela utiliza o *design pattern* *Singleton*. O objetivo desse *design pattern* é definido no livro *“Design Patterns: Elements of Reusable Object-Oriented Software”* (1995) para garantir que existirá apenas uma instância de um determinado objeto e que essa instância estará disponível de forma púbica em todo o escopo da aplicação. Portanto, a classe “ORM/Orm” retém e centraliza todas as informações públicas pertinentes ao ORM, que estarão acessíveis para todo o ORM e no escopo da aplicação.

No entanto, cada banco de dados requer que uma conexão diferente seja estabelecida e como pode-se verificar na Listagem 5, cada conexão também requer informações diferentes. Para solucionar essa questão, é necessário delegar essa tarefa a alguém responsável por conhecer as particularidades de cada banco de dados. A solução para isso é o *Driver*.

## *Drivers*

Os bancos de dados possuem muitas características em comum, porém, existem várias características que, embora tenham o mesmo objetivo, são realizadas de maneiras diferentes. Essas diferenças são centralizadas no *Driver* e isoladas do ORM, portanto, o banco de dados pode ser trocado a qualquer momento sem que o ORM precise passar por qualquer adaptação.

Além das variações de banco de dados, essas informações podem variar de versão para versão do mesmo banco de dados. E por isso é necessário que haja uma implementação para cada banco de dados e suas versões para que assim, o ORM possa lidar com todas as particularidades de cada um, separadamente.

O ORM identifica qual *Driver* deve utilizar, usando as chaves “db”, “version” que se encontram na declaração da conexão no arquivo “connection.config.php” conforme abordado na seção Conexões.

Caso uma versão não seja definida, é possível usar um *Driver* genérico. O *Driver* genérico contém as informações mais comuns do banco de dados em questão, por exemplo, o *Driver* do MySQL na versão 5.7 possui as informações específicas para essa versão, e o *Driver* genérico do MySQL contém as informações mais comuns desse banco de dados.

### **Criar Conexões**

Enquanto o papel do ORM, entre outras coisas, é centralizar as conexões, um dos papeis do *Driver* é criar a conexão com o banco de dados.

O *Driver* valida as informações que constam na declaração da conexão no arquivo “connection.config.php” (conforme abordado na seção Conexões) e utiliza as informações para criar uma conexão e retorna para o ORM.

### **Tipos de Dados**

O *Driver* concentra os tipos de dados que o ORM pode aceitar na propriedade “type” da *annotation* “@ORM/Column” e também os tipos correspondentes para o banco de dados, por exemplo, o tipo “string” que é informado na *annotation* “@ORM/Column” é um tipo de dados aceito pelo ORM e “varchar” é o tipo de dado equivalente para o banco de dados MySQL.

Para o ORM mapear as classes do modelo para executar os comandos nos dados do banco de dados e mapear os resultados oriundos do banco para as classes do modelo, é necessário converter os dados do PHP para o banco de dados e do banco de dados para o PHP. Para isso, o *Driver* contém a lógica para converter os tipos contidos nele. Para os tipos de dados “date”, “datetime” e “time” que requerem um formato específico para ser armazenado no banco de dados, o *Driver* contém também o formato para que o ORM possa formatar os dados no formado correto.

### **Valores Auto Gerados**

O ORM possui a *annotation* “@ORM/Generated”, que indica que a chave primária da tabela que a classe modelo está mapeando é auto gerada. Mas, como cada banco de dados gera o valor da chave primária de maneira diferente, o *Driver* deve indicar ao ORM como o banco de dados trabalha nesse aspecto. Existem 3 maneiras para gerar o valor da chave primária que o ORM pode considerar e o *Driver* tem por obrigação informar o tipo pelo qual o valor será gerado.

O primeiro tipo é indicando na coluna da tabela no banco de dados, por exemplo, no MySQL a propriedade da coluna que é a chave primária chama-se “AUTO\_INCREMENT” e no SQLite o nome da propriedade é “AUTOINCREMENT”. Essa diferença também é indicada pelo *Driver*.

O segundo tipo é conhecido como *sequence*. Uma *sequence* é um objeto que gera um número sequencial e está disponível em bancos de dados como o Oracle e o PostgreSQL. O ORM possui também o nome de uma *sequence* genérica, o qual deve ser utilizado para gerar o valor da chave primária de todas as tabelas mapeadas na aplicação.

E por fim, o terceiro tipo, que em último caso, pode ser usado para gerar o valor da chave primaria através de uma consulta na tabela mapeada pela classe do modelo, buscando maior valor da chave primária da tabela para incrementar manualmente. A query a ser usada é fornecida pelo *Driver*.

### **Paginação**

Em consultas que retornam uma grande quantidade de registros, uma abordagem comum dos desenvolvedores é dividir os registros em páginas ou simplesmente retornar somente os registros mais recentes. Então, quando o ORM necessita criar uma consulta paginada ou trazendo somente os primeiros registros resultantes da consulta, o *Driver* possui as informações necessárias para o ORM montar essa consulta.

## Criação das Tabelas

O ORM tem a habilidade de criar as tabelas a partir das classes modelo. Para que o ORM saiba como criar, é necessário informar o caminho para a pasta que contém os modelos e o *namespace* no momento em que estiver configurando a conexão no ORM.

O *namespace*, na definição mais ampla, é uma forma de encapsular itens, por exemplo, em qualquer sistema operacional, os diretórios servem para agrupar arquivos relacionados, e atuam como um *namespace* para os arquivos, então, no sistema operacional é possível dois arquivos com o mesmo nome coexistirem desde que estejam em pastas diferentes. Esse mesmo princípio se estende a *namespaces* no mundo da programação. *Namespace* no PHP fornece uma maneira de agrupar classes, interfaces, funções e constantes relacionadas.

Então, para configurar o ORM para criar as tabelas, segue o exemplo:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’, [
4. ‘namespace’ => ‘App’,
5. ‘modelsFolder’ => ‘/home/user/app/models’,
6. ‘create’ => true
7. ]);
8. ?>

Listagem 7 – Exemplo para configurar a criação de tabelas

Fonte: Autor, 2018

Pode ser necessário também, apagar as tabelas antes de cria-las, para isso, basta informar também na configuração da conexão:

1. <?php
2. $orm = ORM\*Orm*::getInstance();
3. $orm->setConnection(‘exemplo-mysql’, [
4. ‘namespace’ => ‘App’,
5. ‘modelsFolder’ => ‘/home/user/app/models’,
6. ‘create’ => true,
7. ‘drop’ => true
8. ]);
9. ?>

Listagem 8 – Exemplo para configurar para apagar as tabelas

Fonte: Autor, 2018

O ORM possui todas as informações necessárias para criar as tabelas usando o mapeamento definido nas classes do modelo em conjunto com as informações sobre o banco de dados contidas no *Driver*, como os tipos de dados e como fazer as chaves primárias com valores auto gerados.

Porém, criar e apagar as tabelas no banco de dados não é uma tarefa muito trivial, ela precisa ser executada em ordem.

Para executar a ação de criar, o ORM precisa ler o mapeamento da classe do modelo, e identificar os relacionamentos da classe. O ORM deve criar primeiro as tabelas as quais a tabela mapeada pelo modelo antes de criar a tabela do mapeada pela classe que o ORM está trabalho no momento. Então de forma recursiva, o ORM irá criar o comando de criação de tabela partindo da tabela que não possui nenhuma referência para outras tabelas até a tabela que possui mais referências. Uma vez que a lista de tabelas esteja completa, o ORM irá executar os comandos na ordem em que foram criados.

A ação de apagar as tabelas do banco de dados funciona de forma semelhante, porém, para executar essa tarefa, o ORM irá começar a criar para apagar a tabela partindo da tabela que possui maior número de referências apontando para ela, para a tabela que não possui nenhuma tabela referência apontando para ela. Então, mais uma vez, o ORM irá executar os comandos na ordem em que foram criados.

O ORM permite que uma ação seja executada antes de apagar as tabelas e uma ação após criar as tabelas. Essas ações podem ser úteis para criar uma rotina de *backup/restore* ou de migração de banco de dados. Para informar o ORM quais ações ele deve executar, basta fazer o seguinte:

1. <?php
2. $dbHelper = new App\Helpers\*InitDatabase()*;
3. $orm = ORM\*Orm*::getInstance();
4. $orm->setConnection(‘exemplo-mysql’, [
5. ‘namespace’ => ‘App’,
6. ‘modelsFolder’ => ‘/home/user/app/models’,
7. ‘drop’ => true,
8. ‘create’ => true,
9. ‘beforeDrop’ => [ $dbHelper, ‘beforeDrop’ ],
10. ‘beforeCreate’ => [ $dbHelper, ‘afterCreate’ ]
11. ]);
12. ?>

Listagem 9 – Exemplo para executar ações antes de apagar as tabelas e depois de cria-las

Fonte: Autor, 2018

Na linha 2, é criado uma instância da classe “App\Helpers\InitDatabase” e nas linhas 9 e 10, é informado para a conexão respectivamente, quais métodos devem ser executados antes de apagar as tabelas e depois de criá-las. Utilizando esses métodos é possível que o desenvolvedor desenvolva uma lógica de como realizar o backup das informações essenciais do banco de dados antes de apagar as tabelas e posteriormente restaurar essas informações após a criação das mesmas.

# Codificação

Toda a parte de codificação será anexa na mídia física entregue junto a esta documentação.

O código fonte relacionado ao *framework* Lamberjack’s ORM pode ser encontrado no seguinte repositório do GitHub https://github.com/dfrancklin/orm/

# Considerações Finais

Em desenvolvimento de software, quando um novo projeto é iniciado, é necessário pensar no futuro, pois um dos processos mais caros nos dias de hoje na área, depois da correção de erros, é a manutenção de um software.

Existe ainda uma preocupação extra no que diz respeito a escolha de um banco de dados, pois em uma situação onde o banco de dados não atende as necessidades do sistema, é preciso substituí-lo por um melhor. Porém, sem a utilização de um framework ORM, as aplicações ficam completamente dependentes de um único banco de dados e qualquer tarefa de melhoria ou manutenção pode se tornar muito mais complicada.

No mercado, atualmente, existem opções de framework ORM em PHP, entretanto, a maioria deles tem problemas com documentação ou não atendem as necessidades do sistema, principalmente na questão de mapeamentos N para N, e ainda, não oferecem a possibilidade de criação de tabelas a partir das classes do modelo.

Sendo assim, o objetivo deste trabalho foi atingido com o desenvolvimento de um framework ORM utilizando a linguagem PHP, um framework ORM que possa, de maneira simples, ser utilizado para mapear modelos de dados complexos, suportando todos os tipos de relacionamentos de tabelas do banco de dados e, criar as tabelas no banco de dados seguindo as classes do modelo de dados da aplicação.

Com a facilidade de modificar somente a conexão do banco de dados que o framework ORM utiliza, é possível trocar para outro banco de dados, completamente novo, sem que haja necessidade de adaptar qualquer trecho de código produzido na aplicação. E com apenas algumas linhas, ter a habilidade de criar as tabelas do banco de dados e fazer com que o ORM crie todas as tabelas necessárias para a aplicação. Tendo ainda a possibilidade de executar ações antes e depois da criação das tabelas, por exemplo, para incluir dados em tabelas de configuração ou para backup.

# Referências

**FAYAD, Mohamed; SCMIDT, Douglas; JOHNSON, Ralph.** Building Applications Frameworks. John Willey, 1999.

**FOWLER, Martin.** UML Essencial: Um breve guia para a linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre, Bookman, 2005.

**FOWLER, Martin.** https://www.martinfowler.com/eaaCatalog/activeRecord.html. Última visita em 30 de abril de 2018.

**FOWLER, Martin.** https://martinfowler.com/eaaCatalog/dataMapper.html. Última visita em 30 de abril de 2018.

**FOWLER, Martin.** https://martinfowler.com/eaaCatalog/lazyLoad.html. Última visita em 30 de abril de 2018.

**GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John.** Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

**JONES, Meilir Page.** Fundamentos do desenho Orientado a Objetos com UML. São Paulo, Makron Books, 2001.

**KOENIG, Andrew.** http://wiki.c2.com/?AntiPattern. Última visita em 30 de abril de 2018.

**ALEXANDER, Christopher.** et al. A Pattern Language. Oxford University Press, New York, 1977.

**GUERRA, Eduardo.** Design Pattern com Java: Projeto orientado a objetos guiado por padrões. Casa do Código, 2012.

**PHP.** http://php.net/manual/en/function.preg-match.php. Última visita em 30 de abril de 2018.

**PHP.** http://php.net/manual/en/function.preg-match-all.php. Última visita em 30 de abril de 2018.

# Glossário

**ORM:** Sigla em inglês que significa *Objeto-Relational Mapping*, ou em português “Mapeamento Objeto-Relacional”

**SQL:** Sigla em inglês que significa *Structured Query Language*, ou em português “Linguagem de Consulta Estruturada”, é a linguagem de pesquisa declarativa padrão para banco de dados relacional.

**Framework:** Em desenvolvimento de software, é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica.

**Design Pattern:** Em Engenharia de Software, um *Design Pattern* (ou em português “padrão de projeto”) é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software.

**PHP:** Sigla em inglês *Hypertext Preprocessor*, é uma linguagem interpretada, usada originalmente apenas para o desenvolvimento de aplicações web, atuante no lado do servidor.

**Annotation:** É um recurso para adicionar metadados que podem ser posteriormente interpretadas que irá realizar ou influenciar em alguma tarefa pré-definida.

**Metadados:** São dados sobre outros dados. Um metadado pode indicar informações ou características sobre outro dado.

**GitHub:** Uma plataforma de hospedagem de código-fonte com controle de versão usando o *Git*.

**Git:** Um sistema de controle de versão distribuído e um sistema de gerenciamento de código-fonte.

**View****:** Pode ser definida como uma tabela virtual composta por linhas e colunas de dados vindos de várias tabelas relacionadas.

**Data Mapper:** Um *Desgin Pattern*, nomeado pela primeira vez por Martin Fowler.

**Active Record:** Um *Desgin Pattern*, nomeado pela primeira vez por Martin Fowler.

**Anti-Pattern:** Um *Desgin Pattern* que pode ser comumente utilizado, porém é ineficiente e/ou contra-produtivo em prática.

**Herança:** Um princípio da orientação a objetos, permite que atributos e métodos de classes sejam compartilhados com outras classes. A herança é usada para reaproveitamento de códigos ou especializar operações.

**Polimorfismo:** Do grego que quer dizer várias formas. É uma característica da orientação a objetos que permite que uma classe mais genérica represente uma classe mais específicas que herdam ou implementam a classe mais genérica, assim é possível que vários tipos sejam tratados da mesma maneira.

**Abstração:** é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstração de entidades existentes no domínio do sistema de software.

**Interface:** É um contrato que define um determinado conjunto de métodos que devem ser implementados nas classes que assinam esse contrato.

**Classe Abstrata:** Serve como um modelo para outras classes que dela herdam, e especializam e implementam métodos da classe abstrata.

**Script:** Ou **Linguagem de Script**, é uma linguagem de programação que é interpretada. Um script é uma séria de instruções em código escritas para serem seguidas de maneira sequencial para realizar uma tarefa computacional.

**Open Source:** é um termo em inglês que significa código aberto. Isso diz respeito ao código-fonte de um software, que pode ser adaptado para diferentes fins.

**Reflection:** em português significa Reflexão, é o processo em que um programa pode observar e modificar sua própria estrutura, no PHP este recurso possibilita realizar engenharia reversa em classes, interfaces, funções e extensões, além disso também podemos resgatar comentários/documentação de funções, classes e métodos.

**Driver:** Contém as informações e funções a serem integradas a um sistema para controlar ou integrar com um determinado sistema ou periférico. No caso, permite que o *framework* se comunique corretamente com diferentes bancos de dados.

**Anexo – Manual do Desenvolvedor**

Texto.