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](#_Toc516079749)

[1. Objetivo 2](#_Toc516079750)

[2. Introdução 4](#_Toc516079751)

[3. Framework 5](#_Toc516079752)

[4. Design Patterns 7](#_Toc516079753)

[4.1. Quando não usar? 8](#_Toc516079754)

[4.2. Qual a diferença entre o Design Pattern e o Framework 8](#_Toc516079755)

[5. ORM 9](#_Toc516079756)

[5.1. Active Record 9](#_Toc516079757)

[5.2. Data Mapper 10](#_Toc516079758)

[6. Estudo de caso 11](#_Toc516079759)

[7. Codificação 12](#_Toc516079760)

[8. Considerações Finais 13](#_Toc516079761)

[Referências 14](#_Toc516079762)

[Glossário 15](#_Toc516079763)

**Lista de Figuras**

[Figura 1: Estrutura do Framework 2](#_Toc512939201)

**Lista de Tabelas**

**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. E na criação de novos projetos, exige 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 todas as funcionalidades do *framework* e uma análise de como é facilitada a programação de aplicações utilizando a linguagem de programação PHP e que necessitam de um banco de dados relacional 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 micro-arquiteturais 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 pessado criando, atualizando, deletando e consultando dados do bando de dados.

Em uma linguagem de programação orientada a objetos, o principal ponto de refencia 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 relaciomentos 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

Falar sobre o que foi feito e como foi feito.

# 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

Texto.

# 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.

# 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.

**Anexo – Manual do Desenvolvedor**

Texto.