IV Congresso de Pesquisa e Extensão da FSG
II Salão de Extensão

http://ojs.fsg.br/index.php/pesquisaextensao
ISSN 2318-8014

ARQUITETURA DE SOFTWARE PHP COM FERRAMENTAS OPENSOURCE: PRODUTIVIDADE, BOAS PRÁTICAS E DESIGN PATTERNS

Fábio Brandt Giordani

- Aluno do Curso Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia da Serra Gaúcha

Me. Pedro Augusto Bocchese

- Doutorando em Ciências da Linguagem na UNISUL

Informações de Submissão

pedro@bocchese.com.br
Rua Leonildo Monarini, 160. Antônio Prado-95250-000 - RS
Rua Os Dezoito do Forte, 2366 - Caxias do Sul - RS - CEP: 95020-472

Palavras-chave:

Arquitetura de Software. Design Patterns. Open source. PHP.

Resumo

O presente artigo tem como objetivo uma proposta de arquitetura de software com a linguagem PHP, utilizando o paradigma de orientação a objetos, com uso de ferramentas open source Symfony2 e Doctrine2, buscando melhor qualidade e produtividade através do emprego de boas práticas de programação e alguns dos padrões de projeto de software reconhecidos no mercado, ambos assegurados pelas ferramentas. Para isto, foi realizada uma pesquisa bibliográfica exploratória com abordagem qualitativa para fundamentação de conceitos de arquitetura de software baseada em reuso de componentes, comunidade open source e a definição de alguns dos padrões de projeto mais comuns, possibilitando relacionar este conhecimento com trechos específicos da arquitetura proposta..

1 INTRODUÇÃO

O tempo e a qualidade são pilares que impactam diretamente no sucesso e no custo de qualquer projeto. Existe a necessidade de nos tornarmos mais eficazes no desenvolvimento de sistemas, por isso a importância de aliar a engenharia de software e ferramentas *open source* para que juntas resultem na qualidade, velocidade e redução de custos.

A ABNT, na norma técnica NBR 10006, define projeto como “Processo único, consistindo de um grupo de atividades coordenadas e controladas com datas para início e término, empreendido para alcance de um objetivo conforme requisitos específicos, incluindo limitações de tempo, custo e recursos”. De acordo com o PMI (*Project Management Institute*), projeto é “Um esforço temporário empreendido para criar um produto, serviço ou resultado único.”. (PMBOK, 2013 p. 3).

Segundo Pressman (2011), para que projetos de software sejam bem-sucedidos, é necessário estar atento a análise de alguns parâmetros, como por exemplo, o escopo do software, os riscos envolvidos, os recursos necessários, as tarefas a serem realizadas, os indicadores a serem acompanhados, os esforços e custos aplicados e a sistemática a ser seguida. Tais parâmetros influenciam e são influenciados diretamente pela arquitetura de software, são decisões que se iniciam antes do desenvolvimento da solução e prossegue à medida que a entrega do software vai se concretizando.

Este artigo visa demonstrar a forma com que ferramentas *open source* podem beneficiar projetos de software *web* na linguagem PHP orientado a objetos, agregando qualidade e eficiência por meio de boas práticas de programação e padrões de projeto de software orientado a objetos.

O *framework* Symfony2 e a biblioteca Doctrine2 foram ferramentas *open source* escolhidas, levando em consideração sua popularidade e posicionamento das ferramentas na comunidade *open source* e no mercado. Através da arquitetura de software, uma das disciplinas que a engenharia de software engloba, se torna possível o entendimento de boas práticas e *design patterns* para o relacionamento com as ferramentas propostas, a fim de compreender de que forma estas tecnologias e a comunidade de software livre podem contribuir de forma benéfica aos projetos de software.

2 REFERENCIAL TEÓRICO

2.1 Engenharia De Software

Bauer (1969, p. 231) foi o primeiro dizendo: "Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia a fim de obter software de maneira econômica, que seja confiável e que trabalhe em máquinas reais". O próprio significado de engenharia já traz os conceitos de criação, construção, análise, desenvolvimento e manutenção.

Segundo Sommerville (2011), a engenharia de software aborda a produção de software como um todo, desde seus estágios iniciais de definição até sua manutenção, que ocorre após a implantação e utilização por parte dos usuários. O autor ressalta também dois fatores importantes, que cada vez mais dependemos de sistemas de software avançados para solucionar os problemas atuais, por isso a produção de sistemas precisa ser aperfeiçoada a fim de concebermos sistemas mais confiáveis e econômicos de forma mais eficaz, trazendo benefícios durante o projeto e em seu período de manutenção. A aplicação de técnicas de

engenharia de software ao longo prazo são mais baratas, para o autor é muito mais custoso a manutenção de software depois que está sendo usado.

Pressman (2011) defende que profissionais de software devem desprender esforços para no emprego e boas práticas para produção de sistemas com alta qualidade, no entanto reconhece que a as práticas que agregam qualidade tem seu devido custo em tempo e dinheiro, defende o equilíbrio e ponderação entre os custos relacionados a qualidade e os custos.

Pressman (2011, *apud* Bertrand Meyer, p. 365) afirma que se fizermos sistemas de baixa qualidade não haverá interessados na sua compra, mas por outro lado se for empreendido tempo demasiados em se conceber um software perfeito, isso o tornará muito caro e levará tanto tempo para o projeto finalizar que as oportunidades de mercado podem se esgotar junto com os recursos dos patrocinadores.

De acordo com o gráfico 1, os esforços em estágios iniciais e software são muito menos custosos, a arquitetura de software é considerada como o primeiro estágio no processo de desenvolvimento, porque dá a liga entre os requisitos levantados e os componentes estruturais que se relacionam. Uma das decisões de um projeto de arquitetura é a escolha de padrões, estratégia utilizada para modelar e controlar o funcionamento dos componentes do sistema.

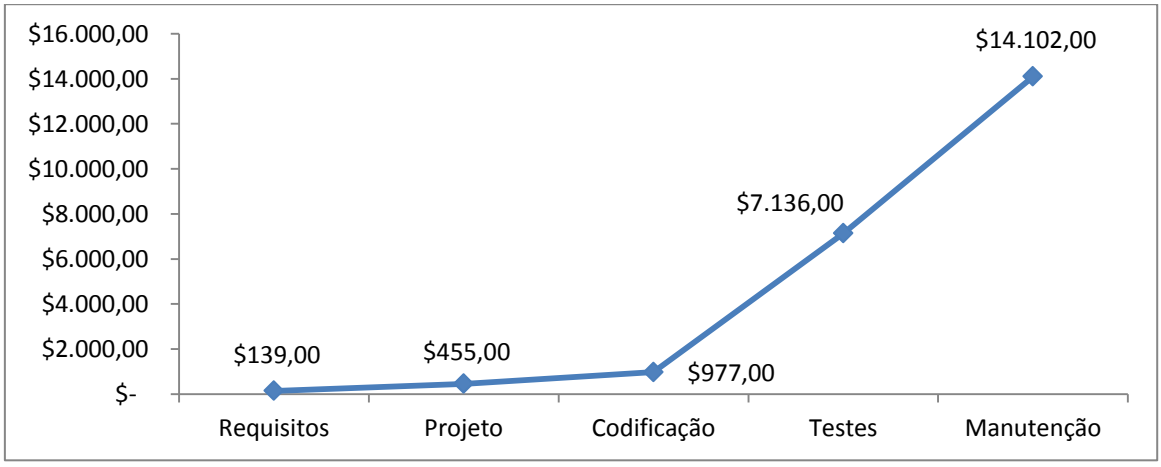

Gráfico 1: Custo relativo para correção de erros e defeitos.
Fonte: Pressman (2011).

A arquitetura baseada em reuso de software é uma estratégia baseada na reutilização de software*s*, segundo Sommerville (2011, *apud* McIlroy, p. 296), uma resposta às exigências do mercado que buscava por menores custos de produção e manutenção, rapidez nas entregas de sistemas com maior qualidade. O reuso é incentivado nas empresas e comunidades para

aumentar o retorno sobre os investimentos em software e disseminação do conhecimento. Com movimento *open source,* a quantidade de software reutilizável aumentou significativamente, a disponibilidade de sistemas e bibliotecas das mais variadas aplicações representa uma base de código disponível a baixo custo, mantidas muitas vezes por empresas que se beneficiam do compartilhamento destes programas.

Segundo Sommerville (2011), dentre os benefícios de software*s* reusados, estão o aumento da confiança, já que sistemas testados e já em ambiente produtivo são mais confiáveis do que os novos software*s*. O reuso permite que o conhecimento e a solução de especialistas, como padrões de projetos e componentes, sejam reutilizados, acelerando o desenvolvimento e reduzindo os custos por consequência.

2.2 Programação Orientada A Objetos

Se tratando de reutilização de código, a Programação Orientada a Objetos (POO) é a abordagem mais difundida para a concepção de sistemas, segundo o portal DEVMEDIA[1] (2014) e Dall'Oglio (2015), a orientação a objetos propicia a reutilização do código criado, diminuindo o tempo de desenvolvimento, bem como o número de linhas de código, sendo isso possível devido ao fato de que a modelagem orientada a objetos objetiva a construção de representações claras e interdependentes dos elementos do software, estruturas que carregam dados e comportamento próprio, além de trocarem mensagens entre si com o objetivo de formar algo maior. Essa independência entre as partes do software é o que permite que esse código seja reutilizado em outros sistemas.

Segundo os autores Martin e O'Doell (1996) e Bahrami (1999), tal popularidade se dá ao fato de promover aspectos como a facilidade de reutilização de código, adição de novos recursos ao sistema, manutenibilidade e aumento da qualidade, possibilitando que modelos de negócio reflitam o mundo real de forma mais precisa. A soma dessas vantagens técnicas acaba refletindo financeiramente, pois o custo de implementação, aprimoramento e manutenção é reduzido.

2.3 Padrões De Projeto

Com a popularização da POO, soluções para problemas de *design* comuns entre projetos começaram a ser catalogados em *design patterns*, padrões de projeto, segundo

[1] Portal DEVMEDIA. Disponível em: http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264 Acesso em: 10 jun. 2016.

Gamma *et al.* (2005, p. 20), Padrões de Projeto definem-se por: "descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular.". Padrões de projeto vêm também com a motivação de facilitar o entendimento entre as esquipes, compartilhamento de experiência entre projetistas experientes e desenvolvedores novatos e promover as vantagens que a orientação a objetos provê.

2.3.1 Padrões De Criação

Segundo Gamma *et al.* (2005), os padrões de criação se propõem a abstrair o processo de instanciação de objetos, pois ajudam a tornar um sistema independente de como seus estes são criados, compostos e representados.

A importância destes padrões aumenta à medida que os sistemas tornam-se mais dependentes da composição de objetos do que a herança de classes, desta maneira o sistema não precisa se preocupar com questões sobre, como o objeto é criado, como é composto, qual a sua representação real. Quando um sistema não precisa se preocupar com a instanciação dos objetos, significa que quando ocorre uma mudança neste ponto a aplicação não é afetada, agradando flexibilidade. Gamma *et al.* (2005), catalogou os padrões de criação com as seguintes definições:

- *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 complexo 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 deixar as subclasses decidirem qual classe ser instanciada. O *Factory Method* permite uma classe postergar (*defer*) a instanciação às subclasses.";
- *Prototype*: "Especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando esse protótipo.";
- *Singleton*: "Garante que uma classe tenha somente uma instancia e fornece um ponto global de acesso para ela.".

2.3.2 Padrões Estruturais

Segundo Gamma *et al.* (2005), os padrões estruturais se preocupam com a forma como classes e objetos são compostos para criar estruturas maiores, ou seja, eles são usados para capturar intenções focadas na composição estrutural de classes e objetos.

Os padrões de classe, através da herança, propõem a composição de interfaces ou implementações, já os padrões de objeto, descrevem formas de composição dos mesmos para obtenção de novas funcionalidades, tendo como ganho a flexibilidade na composição de objetos em tempo de execução. Gamma *et al.* (2005), catalogou os padrões de estrutura com as seguintes definições:

- *Adapter*: "Converte a interface de uma classe para outra interface esperada pelos clientes. O *Adapter* permite que classes com interfaces trabalhem em conjunto, pois de outra forma seria impossível por causa das suas interfaces incompatíveis.";
- *Bridge*: "Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.";
- *Composite*: "Compõe objetos em estruturas de árvore para representar hierarquias partes-todo. O *Composite* permite aos clientes tratarem objetos individuais e composições de objetos de maneira uniforme.";
- *Decorator*: "Atribui reponsabilidades adicionais a um objeto dinamicamente. Os *Decorators* fornecem uma alternativa flexível ao uso de subclasses para extensão da funcionalidade.";
- *Facade*: "Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O *Facade* 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.".

2.3.3 Padrões Comportamentais

Segundo Gamma *et al.* (2005), os padrões comportamentais se preocupam com algoritmos e atribuições de responsabilidade entre objetos, classes e os padrões de comunicação que podem haver entre eles.

Padrões de comportamento de classe utilizam a herança para definir a estrutura e o comportamento entre classes, enquanto os padrões comportamentais de objetos utilizam a

composição, a combinação de padrões comportamentais de objetos e classes, podem ser considerados como um grupo de objetos que cooperam entre si executando uma tarefa que nenhum deles conseguiria desempenhar sozinho. Gamma *et al.* (2005), catalogou os padrões de comportamento com as seguintes definições:

- *Chain of Responsibility*: "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*) de solicitações e suporte operações que podem ser desfeitas.";

- *Interpreter*: "Dada uma linguagem, define uma representação para a sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem.";

- *Iterator*: "Fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objetos sem expor sua representação subjacente.";

- *Mediator*: "Define um objeto que define a forma com que um conjunto de objetos interagem. 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 o encapsulamento, 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 forma que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.";

- *State*: "Permite que um objeto mude 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 torna intercambiáveis. O *Strategy* permite que o algoritmo varie independente dos clientes que o utilizam.";

- *Template Method*: "Define um 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.".

2.3.4 Padrões de Arquitetura de Aplicações Corporativas

Fowler *et al.* (2002) também contribuiu com a catalogação de padrões de projeto, junto de um grupo de colaboradores, elencou padrões que consideravam mais comuns na arquitetura de aplicações corporativas, uma resposta aos problemas mais comuns da época. Os padrões citados e catalogados ainda hoje se fazem presentes em projetos de software. Alguns destes padrões são:

- *Front Controller*: O Front Controller consolida todas as requisições e as canaliza através de um único objeto manipulador;
- MVC: Separa as camadas de informação da interação entre visão modelo e controle da aplicação;
- *UnityOfWork*: "Mantém uma lista de objetos afetados por uma transação do negócio e coordena a escrita de mudanças e a resolução de problemas de concorrência.";
- *Request-Response*: Padrão de troca de mensagens comuns em aplicações;
- *Dependency Injection*: Padrão que possibilita que as dependências de determinado elemento, como por exemplo, uma classe, sejam injetadas em tempo de execução, agregando flexibilidade ao código. Tal injeção pode ser feita via construtores de classe, chamadas de métodos e definição de propriedades;
- *Repository*: Camada que media regras de negócio e persistência, nesta camada ficam as ações de consulta ao bando de dados;
- *Template View*: "Processa informações em HTML por meio de marcadores de incorporação em uma página HTML".

2.4 PHP

De acordo com a Zend (2007), o PHP conta com mais de dez anos de constante desenvolvimento, milhões de desenvolvedores utilizam a linguagem em tempo integral para desenvolver sistemas, somado mais de 20 milhões de sites e aplicações, hoje o PHP é uma linguagem abrangente e repleta de recursos de programação com sólido suporte a orientação a objetos. Outro benefício do PHP é a flexibilidade, uma vez que não é necessário ser

compilado, facilitando o *deploy* de novas versões, alterações ou correções de erros em poucos minutos. A prototipação de conceitos e aplicações também é facilitada, normalmente levando em torno de 50% do tempo em comparação com C ou Java.

Para Dall'Oglio (2015), a popularidade da linguagem se deve ao fato da facilidade de se criar soluções por meio de estruturas de programação flexíveis e suporte a maioria dos bancos de dados existentes no mercado de dados. Viabilizando a criação de *websites*, portais e até aplicações de negócio de grande complexidade, graças ao emprego de OO e boas práticas.

Segundo a pesquisa feita pela RedMonk[2] divulgada em junho de 2015 o PHP atualmente é a terceira linguagem mais popular do mundo, ficando atrás do Java e JavaScript, a pesquisa teve base nas discussões sobre linguagem e *posts* relacionados as mesmas, publicados no site StackOverflow, além das métricas de utilização da linguagem no GitHub, maior serviço de hospedagem de repositórios de código do mundo com aproximadamente 3,4 milhões de usuários[3]. As duas fontes foram utilizadas devido a sua popularidade e utilização na comunidade de desenvolvedores.

De acordo com o portal Packagist[4], mais de dois milhões de pacotes públicos foram registrados desde 13 de abril de 2012, este dado demonstra a popularidade da linguagem e a força da comunidade *open source* no compartilhamento de componentes e soluções de software para reuso. Zend (2007) afirma também que PHP é recomendado não só pela sua grande comunidade *open source*, mas também pelos principais profissionais do mercado de TI como IBM, Oracle e Microsoft. Ressalta também a vantagem do PHP ter independência de plataformas, podendo ser hospedado em diferentes sistemas operacionais e arquiteturas.

Por ser uma linguagem *open source*, as boas práticas de codificação estão espalhadas em diversos canais de informação da *Web*, sendo o PHP-FIG (PHP *Framework Interop Group*) um dos mais relevantes, pois consiste em um grupo de representantes dos maiores projetos em PHP do mercado, com o objetivo de discutir e formalizar boas práticas junto da comunidade, para que projetos possam trabalhar juntos deforma harmônica e padronizada. Além das recomendações definidas, promove movimentos como PHP *The Right Way* (2016), site também mantido pela comunidade que agrupa informações acerca da linguagem, como boas práticas, padrões, segurança, documentação, etc.

[2] Portal RedMonk. Disponível em: http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15 Acesso em: 10 jun. 2016.
[3] Portal GitHut. Disponível em: http://githut.info Acesso em: 10 jun. 2016.
[4] Portal Packagist. Disponível em: https://packagist.org/statistics Acesso em: 10 jun. 2016.

2.5 Symfony2

Symfony2 é um *framework* PHP *fullstack* (completo), reconhecido internacionalmente. O *framework* assegura que a aplicação está em conformidade com as regras da indústria como estar bem estruturado, manutenível e escalável. Também economiza tempo pela reutilização de módulos, permitindo que os profissionais que o utilizam possam focar mais tempo nas características do negócio.

Symfony2 foi criado pela empresa SensioLabs, um dos produtos gratuitos da empresa que contribui ativamente na comunidade *open source*. Symfony2 é *open source* porque a empresa acredita que a ferramenta beneficia outros desenvolvedores e que estes têm a possibilidade de melhorá-lo, adicionando novos módulos e contribuindo para a melhoria do produto junto da empresa mantenedora. Isto feito em um ambiente controlado pela equipe de moderadores que asseguram o emprego de boas práticas e padrões de projeto, juntamente de ferramentas automatizadas de testes unitários e integração contínua. Symfony2 faz parte do PHP-FIG e implementa padrões definidos na PSR-0, PSR-1, PSR-2 e PSR-4, convenções abordadas pelo *PHP The Right Way* e outras boas práticas promovidas pela própria documentação[5].

O projeto já conta com mais de duas mil contribuições feitas por mais de trezentos mil desenvolvedores e em torno de cinco milhões de *downloads* mensais. Muitas das grandes aplicações em PHP do mercado se beneficiam dos componentes do *framework*, tais como *Drupal, phpBB, Magento, eZ Publish, Joomla* e inclusive outros *frameworks* do mercado, como o Laravel e Silex[6].

2.6 Doctrine2

O projeto Doctrine se intitula como a casa de um conjunto selecionado de bibliotecas PHP, primariamente focada no fornecimento de serviços de persistência e funcionalidade relacionadas, tendo como seus principais projetos o ORM (*Object Relational Mapper*) e o DBAL (*Database Abstraction Layer*), base do ORM[7].

[5] Portal Symfony. Disponível em: http://symfony.com/doc/current/contributing/code/standards.html Acesso em: 10 jun. 2016.
[6] Portal Symfony. Disponível em: http://symfony.com/projects Acesso em: 10 jun. 2016.
[7] Portal Doctrine. Disponível em: http://www.doctrine-project.org/about.html Acesso em: 10 jun. 2016.

O Mapeador de objeto relacional (ORM) fica no topo de uma camada de abstração de banco de dados poderosa (DBAL). Uma das suas principais características é a opção de escrever consultas de banco de dados em um dialeto SQL orientado a objetos proprietária, chamada DQL (*Doctrine Query Language*), inspirado na *Hibernates* HQL (Java). Isto fornece aos desenvolvedores uma alternativa poderosa ao SQL, já que mantém a flexibilidade sem a necessidade de duplicação de código desnecessário.

Desde 2006, a ferramenta possui estabilidade e qualidade garantida pela comunidade *open source*, suportando diversos casos de uso e integrações com frameworks renomados e amplamente utilizados no mercado de software, entre eles Symfony2, *Zend Framework, CodeIgniter*, etc. O projeto Doctrine também promove boas práticas sugeridas pelo PHP-FIG e relacionadas no *PHP The Right Way*, como versionamento, integração contínua, testes unitários e padrões de codificação[8].

3 METODOLOGIA

O desenvolvimento da primeira etapa deste artigo foi realizado através pesquisa bibliográfica exploratória com abordagem qualitativa, abordada por Gil (2012), coletando dados em livros, pesquisas acadêmicas, artigos científicos e visitas em portais de tecnologia em busca de informações que elucidem o tema proposto, desde a importância da otimização na elaboração e desenvolvimento de projetos de software, engenharia de software como um todo, passando pelos aspectos de qualidade, arquitetura e reuso, este sendo relacionado com a comunidade *open source* e padrões de projeto.

Para a segunda etapa, foi realizada uma análise de arquitetura e sinergia das ferramentas Symfony2 e Doctrine2, identificando alguns dos padrões de projetos que as ferramentas empregam e os elementos que caracterizam a qualidade das ferramentas durante seu fluxo de informações. Foi apresentada uma proposta de arquitetura para desenvolvimento de software orientado a objetos na linguagem PHP na versão 5.6, utilizando o *framework* PHP Symfony2 na versão 2.7 e a biblioteca Doctrine2 (ORM e DBAL) versão 2.5.

A amostragem da arquitetura limita-se a representação do fluxo de informações das requisições envolvidas no processo de edição de uma entidade de usuário, expondo elementos das ferramentas e da arquitetura que exemplifiquem alguns padrões de projeto, portanto, a estrutura de diretórios, versionamento, gerenciamento de pacotes, testes unitários e

[8] Instruções de contribuição Doctrine. Disponível em: https://github.com/doctrine/doctrine2/blob/master/CONTRIBUTING.md Acesso em: 10 jun. 2016.

funcionais, diagramação, modelagem, banco de dados utilizado, bem como estudos aprofundados do framework, manuais e pré-requisitos de instalação não são abordados no artigo.

Foram extraídos trechos específicos da codificação da funcionalidade e elaborados diagramas que representam o fluxo de dados e as camadas arquiteturais do sistema para relacionar com alguns dos padrões de projeto e boas práticas. O fluxo parte do princípio da requisição para editar um formulário de entidade de usuário do sistema, será acompanhado o ciclo de vida desta requisição de sua origem, o processamento pelas camadas da aplicação até a persistência e o caminho de volta, que compreende o momento da persistência no banco de dados até a mensagem de retorno ao usuário.

4 PROPOSTA DE ARQUITETURA

A arquitetura pode ser representada visualmente pelo Diagrama 1 que mostra as camadas do *framework* Symfony2 em harmonia com as bibliotecas Doctrine2. O diagrama elucida o fluxo de informações entre os componentes do *framework* e da biblioteca, relacionando alguns dos padrões de projetos mapeados.

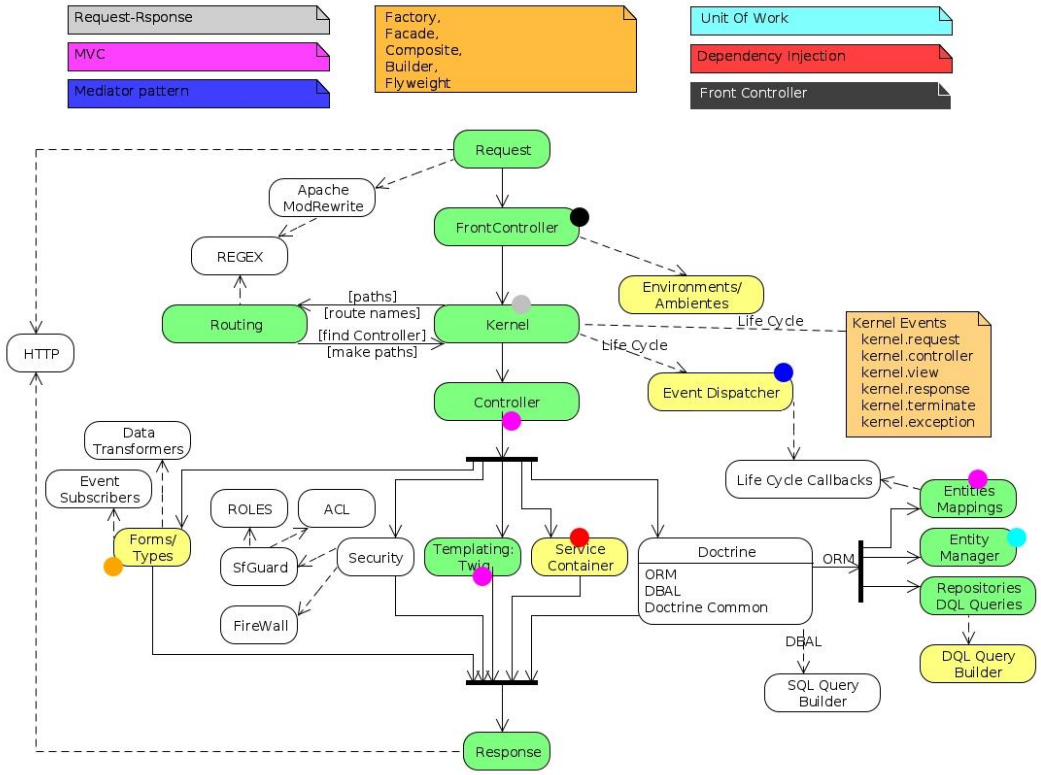

Diagrama 1: Arquitetura Symfony2 e Doctrine2.
Fonte: Adaptado de Stack Overflow (2013).[9]

[9] Portal Stack Overflow (2013). Disponível em: http://stackoverflow.com/a/18791979. Acesso em: 10 jun. 2016.

Observa-se que a requisição é enviada ao FrontConroller, que inicializa o Kernel da aplicação de acordo com o ambiente e suas configurações, este trata a requisição utilizando o componente de rotas do *framework* (Routing Component), que busca as configurações de rota, que está associada a um controlador, de acordo com a URL da requisição. Na arquitetura proposta, a rota é definida no Controller da entidade, elemento fundamental da arquitetura, já que nele temos acesso à requisição e dele parte a resposta (Response). O Controlador da entidade de usuário está representado na figura Figura 1, com o método de atualização da entidade, objeto proposto no artigo.

```php
<?php

namespace UserBundle\Controller\Security;

use AppBundle\Form\Security\UserType;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

/**
 * User controller.
 *
 * @Security("user.isAdmin()")
 * @Route("/admin/user")
 *
 * @author Fábio B. Giordani <fbgiordani@gmail.com>
 */
class UserController extends Controller
{
...
    /**
     * Edits an existing User entity.
     *
     * @Route("/{id}/update", name="admin_user_update")
     * @Method("POST")
     */
    public function updateAction(Request $request, $id)
    {
        $um     = $this->get('app.user.user_manager');
        $entity = $um->find($id);

        $editForm = $this->createForm(new UserType(), $entity);
        $editForm->handleRequest($request);

        if ($editForm->isValid()) {
            $um->persist($entity);

            return $this->redirectToRoute('admin_user');
        }

        return $this->render('admin\user\edit.html.twig', [
            'entity' => $entity,
            'form'   => $editForm->createView(),
        ]);
    }
...
```

Figura 1: UserController, Trecho da classe controladora da entidade de usuários.
Fonte: Autores (2016).

A rota é configurada pela *annotation*[10] @Route, linha 16, que neste caso está definindo que UserController é responsável por tratar requisições cuja URL é, a partir do nome de domínio, "/admin/user". Pode ser observado também na linha 15 da mesma figura, a definição de segurança que garante acesso à classe apenas a usuários (APÊNDICE A) do sistema que possuem perfil de administrador, verificação que acontece no método *isAdmin*, definido na entidade de usuário (APÊNDICE A, linha 479). A entidade de usuário, implementa o padrão Decorator, por estender as interfaces UserInterface e Serializable, também é possível identificar as *annotations* @ORM do Doctrine2, estas configurações determinam como a entidade e atributos são mapeados no banco de dados.

Nas linhas 52 e 53 da Figura 1, pode ser visualizada a definição da rota para edição de usuário, esta rota é combinada com a rota definida no Controller. A configuração expressada por ambas as linhas, se resume na regra que determina que o método *updateAction* seja responsável por tratar requisições POST cuja URL combina com o padrão "/admin/user/{id}/update-user", onde "{id}" é um *placeholder*, que será devidamente substituído pelo identificador da entidade de usuário enviado na requisição. Uma vez que a requisição bate com as regras definidas, o método *updateAction*, é invocado e o objeto da requisição é passado por parâmetro junto do identificador de usuário, variável definida na criação da rota.

Na linha 57 da mesma figura, ocorre a obtenção do UserManager objeto que contém as regras de negócio e são agrupados objetos auxiliares da entidade, como o seu repositório (Padrão *Repository*) e onde são injetadas as dependências (EntityManager e Classe de usuário), sua configuração é representada na Figura 2, que representa um trecho do arquivo de configuração de serviços do Symfony2 em formato YAML[11].

```
 6  services:
 7  #    service_name:
28      app.user.user_manager:
29          class: UserBundle\Entity\Manager\UserManager
30          arguments: ["@doctrine.orm.entity_manager", "UserBundle\Entity\User"]
```

Figura 2: Definição e configuração do UserManager
Fonte: Autores (2016)

[10] No Symfony2 é possível definir rotas e outras configurações por *annotations*, uma das formas de configuração que o *framework* possibilita.

[11] YAML é um formato amigável de serialização de dados: http://yaml.org/ Acesso em: 10 jun. 2016.

Através do UserManager, cujo a classe base pode ser visualizada no APÊNDICE B e APÊNDICE C, é obtido a entidade do usuário, através do método *find*, invocado na linha 58 da Figura 1 e definido na linha 106 do APÊNDICE C, que recebe o id da entidade por parâmetro e retorna o objeto Proxy de usuário, graças ao Doctrine ORM. No método *find* podem ser observadas tratativas de erro no caso da entidade não ser encontrada e a possibilidade de verificar alguma permissão específica, flexibilidades importantes em termos de arquitetura, já que o objetivo do manager é conter a regra de negócio.

Na linha 60 da Figura 1 pode ser observada a criação do objeto de formulário por uma *Factory* de formulários do *framework*, a classe UserType representada parcialmente no APÊNDICE D, utiliza dos padrões *Builder* e *Composite* para compor o formulário, associando os dados da entidade enviada por parâmetro com as configurações de composição da classe, resultado em um objeto de formulário composto e configurado para a entidade em uso.

Após o formulário ser composto, na linha 61 inicia o processo interno do *framework* de tratar a requisição (POST), os dados da requisição são carregados no formulário e este popula novamente a entidade. Neste processo, o dado da requisição, texto plano, é normalizado e convertido para o dado esperado na entidade, a validação ocorre em seguida e de acordo com as *assertions* que podem ser definidas nos atributos da entidade, junto do mapeamento ORM ou definidas na classe de formulário[12].

Em caso de sucesso na validação, a entidade é enviada ao UserManager através do método *persist*, linha 58 do APÊNDICE C, este por sua vez faz uso do EntityManager, recurso do Doctrine2 ORM, injetado como dependência no construtor do UserManager (APÊNDICE B, linha 34). O EntyManager registra na UnityOfWork a entidade a ser persistida no bando de dados e em seguida invoca o *flush*, método onde a conexão ao banco de dados é aberta e as operações registradas são executadas. Após a persistência, o Controller redireciona o usuário para outra rota, responsável por exibir a listagem de usuários.

Em caso de insucesso, é enviado o formulário e entidade e objeto de formulário para que o *template engine Twig*, componente do Symfony2, interprete as variáveis novamente, renderizando a *template* parcialmente definida no APÊNDICE E. No momento da renderização, o componente responsável pela tarefa possui acesso aos objetos, portanto compõe o HTML final com as informações atualizadas, sejam elas dados enviados na

[12] Componente de validação Symfony2: Disponível em: http://symfony.com/doc/current/book/validation.html Acesso 15 jun. 2016

requisição ou mensagens e informações adicionadas por algum componente, como o de validação. Importante ressaltar que o Symfony2 possui sistema de cache para ambiente de produção, desta forma, *templates*, *annotations* e dentre outras configurações são compiladas e armazenadas, melhorando o desempenho em ambiente de produção.

5 CONSIDERAÇÕES FINAIS

Bibliotecas e *frameworks open source* tendem a implementar padrões de projeto, já que são suportados por uma comunidade de desenvolvedores e muitas vezes aparadas por grandes empresas de software e serviços de tecnologia.

Foi possível através de este artigo relacionar alguns dos padrões de projeto reconhecidos pelo mercado com as ferramentas propostas, demonstrando que mesmo soluções *open source* podem agregar a qualidade e maturidade às soluções, já que são muitas vezes amparadas por grandes empresas e grande número de interessados da comunidade. A própria comunidade faz com que os padrões de projeto e qualidade sejam respeitados e priorizados, com o objetivo do reuso de software a benefício de todos. No mercado de software, reinventar a roda significa custo, e a colaboração se mostra a saída para que todos possam se beneficiar através da propagação do conhecimento e compartilhamento de soluções, para que o tempo seja desprendido muito mais no negócio do que nos fundamentos do software.

Este artigo limitou-se a exemplificar as aplicações das bibliotecas Symfony2 e Doctrine2 ORM em uma arquitetura PHP orientada a objetos. Não foram abordados os tipos de licença de software, possíveis problemas com o reuso, *antipatterns*, *code-smelss* e princípios de design como KISS, DRY e SOLID, recomendados para pesquisas, assim como as referências utilizadas, por abordarem o assunto de forma mais aprofundada e ampla. Padrões de projetos e boas práticas não se limitam as abordadas pelo artigo, recomenda-se também a pesquisa de tais assuntos em outras fontes de pesquisa.

6 REFERÊNCIAS

ABNT. **Associação Brasileira de Normas Técnicas. NBR ISO/IEC 12207: Tecnologia de informação:** Processos de ciclo de vida de software. Rio de Janeiro: ABNT, 1998.

BAHRAMI, A. **Object Oriented Systems Development -** using the unified modeling language. [S.l.]: McGraw-Hill, 1999.

BAUER, F L. **Engenharia de Software**. Brussels: NATO Scientific Affairs Division, 1969.

DALL'OGLIO, P. **PHP:** Programando com Orientação a Objetos. São Paulo: Novatec, 2015.

FOWLER, M. **Patterns of Enterprise Application Architecture.** 1. ed. [S.l.]: Addison Wesley, 2002.

FOWLER, M. et al., **UML Essencial: Um Breve Guia para Linguagem Padrão**, 3. ed. Porto Alegre: Bookman, 2011.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Padrões de Projeto: Soluções Reutilizáveis De Software Orientado A Objetos**. Porto Alegre: Bookman, 2005.

GIL, A. C. **Como elaborar projetos de pesquisa**. São Paulo: Atlas, 2002.

MARTIN, J.; O'DELL, J. **Análise e projeto Orientados a Objeto**. Makron Books, 1996.

PMI. **Project Management Institute, Um Guia para o Corpo de Conhecimentos em Gerenciamento de Projetos**. (Guia PMBOK). 5. ed. [S.l]: Global Standard, 2013.

PRESMANN, R. S. **Engenharia de Software: uma abordagem profissional**. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de Software**. São Paulo: Pearson. 2011.

PHP-FIG - PHP FRAMEWORK INTEROP GROUP. Disponível em: <www.php-fig.org>. Acesso em: 13 jun. 2016.

PHP: THE RIGHT WAY. Disponível em: <www.phptherightway.com/>. Acesso em: 13 jun. 2016.

ZEND. **Zend Whitepaper PHP:** An overview on PHP. [S.l.]: Zend Corporation. 2007.

APÊNDICE A - Trecho da classe User, representa a entidade de usuário do sistema.

```php
<?php

namespace UserBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Table(name="security_users")
 * @ORM\Entity(repositoryClass="UserBundle\Entity\Repository\UserRepository")
 * @ORM\HasLifecycleCallbacks
 *
 * @author Fábio B. Giordani <fbgiordani@gmail.com>
 */
class User implements UserInterface, \Serializable
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     *
     * @var integer
     */
    private $id;
...
    /**
     * @var string
     *
     * @ORM\Column(name="name", type="string", length=255, unique=true)
     */
    private $name;
...
    /**
     * Checks if the user is a administrator
     *
     * @return boolean
     */
    public function isAdmin()
    {
        if (null === $this->isAdmin) {
            $this->isAdmin = $this->hasRole(self::ROLE_ADMIN, false);
        }

        return $this->isAdmin;
    }
```

APÊNDICE B - Trecho da classe AbstractManager, classe abstrata que é extendida pelos *managers* de entidade, como UserManager.

```php
<?php

namespace AppBundle\Base\Entity\Manager;

use AppBundle\Exception\MethodNotImplementedException;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\EntityRepository;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @author Fábio B. Giordani <fbgiordani@gmail.com>
 */
abstract class AbstractManager
{
    /**
     * @var EntityManager
     */
    protected $em;

    /**
     * FQCN of the entity controlled by the manager.
     *
     * @var string
     */
    protected $class;

    /**
     * @var EntityRepository
     */
    protected $repository;

    public function __construct(EntityManager $em,  $class)
    {
        $this->em = $em;
        $this->repository = $em->getRepository($class);

        $metadata = $em->getClassMetadata($class);
        $this->class = $metadata->name;
    }

    /**
     * @return string
     */
    public function getClass()
    {
        return $this->class;
    }

```

APÊNDICE C – Trecho da classe AbstractManager.

```
51      /**
52       * @return EntityRepository
53       */
54      public function getRepository()
55      {
56          return $this->repository;
57      }
...
72      /**
73       * Persists the given entity.
74       *
75       * @param object $entity
76       * @param bool   $andFlush Whether to flush the changes (default true).
77       */
78      public function persist($entity, $andFlush = true)
79      {
80          $this->em->persist($entity);
81          if ($andFlush) {
82              $this->em->flush();
83          }
84      }
...
97       /**
98        * Finds an entity by its primary key / identifier.
99        *
100       * @param  mixed            $id              The identifier.
101       * @param  UserInterface    $currentUser     Logged User.
102       * @param  boolean $throwException  Throws an exception when entity is not found (default true).
103       *
104       * @return object
105       */
106      public function find($id, UserInterface $currentUser = null, $throwException = true)
107      {
108          $entity = $this->repository->find($id);
109          if (!$entity && $throwException) {
110              throw new NotFoundHttpException('Unable to find '. $this->class .' entity.', null);
111          }
112
113          if (null !== $currentUser) {
114              $this->verifyPermission($entity, $currentUser);
115          }
116
117          return $entity;
118      }
```

APÊNDICE D – Trecho da classe de formulário UserType.

```php
<?php

namespace UserBundle\Form;

use Doctrine\ORM\EntityRepository;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

/**
 * @author Fábio B. Giordani <fbgiordani@gmail.com>
 */
class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $groupRoles = $options['data']->getAllGroupRoles();

        $builder
            ->add('name', null, array(
                'label' => 'Nome',
                'disabled' => true,
            ))
            ->add('groupRoles', 'choice', array(
                'label'    => 'Permissões herdadas pelos Grupos',
                'mapped'   => false,
                'expanded' => true,
                'multiple' => true,
                'disabled' => true,
                'data'     => array_keys($groupRoles),
                'choices'  => $groupRoles,
            ))
            ->add('roles', null, array(
                'class'         => 'UserBundle:Role',
                'label'         => 'Permissões',
                'property_path' => 'rolesCollection',
                'expanded'      => true,
                'multiple'      => true,
                'query_builder' => function(EntityRepository $er) use ($groupRoles) {
                    $qb = $er->createQueryBuilder('e');

                    $qb->where('e.id NOT IN (' . join(',', array_keys($groupRoles)) .')');

                    return $qb->orderBy('e.name');
                },
            ))
        ;
```

APÊNDICE E – Trecho da *template* de formulário para edição de entidades de usuário.

```
{% extends 'base.html.twig' %}

{% block pageTitle %}Usuários{% endblock %}

{% block pageName %}
    Administração <small> Usuários</small>
{% endblock pageName %}

{% block mainContent %}
    <form action="{{ path('admin_user_update', { 'id' : entity.id }) }}" method="post">
    <div class="box box-success">
        <div class="box-body">
            <div class="row">
                <div class="col-md-3">
                    {{ form_row(form.name) }}
                </div>
            </div>
            <div class="row">
                <div class="col-md-6">
                    {{ form_row(form.roles) }}
                </div>
                <div class="col-md-6">
                    {{ form_row(form.groupRoles) }}
                </div>
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-md-12">
            <div class="box-footer">
                <a href="{{ path('admin_user') }}" class="btn btn-primary">Voltar</a>
                <button class="btn btn btn-success pull-right" type="submit">Salvar</button>
            </div>
        </div>
    </div>
    {{ form_widget(form._token) }}
</form>
{% endblock %}
```