CENTRO UNIVERSITÁRIO DE ADAMANTINA

ESTUDO DE PADRONIZAÇÃO NO DESENVOLVIMENTO WEB COM A LINGUAGEM PHP

ANA PAULA MESSINA

Tecnólogo em Análise e Desenvolvimento de Sistemas

ADAMANTINA – SP
2016

ANA PAULA MESSINA

ESTUDO DE PADRONIZAÇÃO NO DESENVOLVIMENTO WEB COM A LINGUAGEM PHP

Trabalho acadêmico apresentado para o Departamento de Tecnologia em Análise e Desenvolvimento de Sistemas como requisito para a conclusão do curso Tecnólogo em Análise e Desenvolvimento de Sistemas sob orientação do Prof. Dr. José Luiz Vieira de Oliveira.

ADAMANTINA – SP
2016

ANA PAULA MESSINA

ESTUDO DE PADRONIZAÇÃO NO DESENVOLVIMENTO WEB COM A LINGUAGEM PHP

Folha de Aprovação

Adamantina, 14 de Dezembro de 2016.

Assinaturas

____

Orientador: Prof. Dr. José Luiz Vieira de Oliveira

____

Examinador: Prof. Me. André Mendes Garcia

____

Examinador: Prof. Welinton Aparecido Ozelin

Agradecimentos

Gostaria de agradecer primeiramente a Deus, a todos que me ajudaram na realização desta monografia, aos meus pais Pedro e Lúcia, aos professores da UniFAI, principalmente ao meu orientador Prof. Dr. José Luiz Vieira de Oliveira e a Profa. Me. Eliane Vendramini de Oliveira, e especialmente ao meu fiel companheiro Renan de Freitas Cavalieri.

Lista de Figuras

Figura 1. Uma arquitetura genérica em camadas ... 16
Figura 2. Distribuição de esforço de manutenção ... 17
Figura 3. A organização do MVC ... 20
Figura 4. O fluxo MVC com as rotas inclusas ... 21
Figura 5. Exemplo de padrão Estratégia ... 22
Figura 6. Exemplo de padrão Fachada ... 23
Figura 7. Estrutura hierárquica dos arquivos - Estrutural X MVC ... 43
Figura 8. Documentação do sistema estrutural ... 49
Figura 9. Documentação do sistema MVC ... 51
Figura 10. Diagrama de classes do sistema MVC ... 58

Lista de Quadros

Quadro 1. Exemplo de classe e instanciação do objeto 27
Quadro 2. Exemplo de métodos 28
Quadro 3. Exemplo de herança - Classe Pai 29
Quadro 4. Exemplo de herança - Classe Filha 30
Quadro 5. Exemplo de classe abstrata 31
Quadro 6. Exemplo de interface 31
Quadro 7. Exemplo de implementação da interface 32
Quadro 8. Exemplo de polimorfismo utilizando interface 33
Quadro 9. Exemplo de polimorfismo - Classe Jato 33
Quadro 10. Exemplo de polimorfismo – Classe Carro 34
Quadro 11. Código seguindo a norma PSR-2 36
Quadro 12. Exemplo da norma PSR-3 LoggerInterface 36
Quadro 13. Exemplo da norma PSR-4 Autoloaders 37
Quadro 14. Sintaxe para executar o PHPDocumentor 38
Quadro 15. Exemplo de documentação no código utilizando o PHPDocumentor 38
Quadro 16. Trecho de código com as ações do sistema estrutural 43
Quadro 17. Trecho de código da interface de ações do sistema MVC 44
Quadro 18. Trecho de código demonstrando a chamada de arquivos 44
Quadro 19. Trecho de código com método de construir rotas 45
Quadro 20. Trecho de código com método que instancia um controller 45
Quadro 21. Trecho de código com método da ficha cadastral de autores 46
Quadro 22. Trecho de código com método que lista autores pelo código 46
Quadro 23. Trecho de código em HTML com as rotas inclusas 46
Quadro 24. Configuração para troca de componentes 47
Quadro 25. Configuração para troca de repositório 48
Quadro 26. Função comentada do sistema estrutural. 49
Quadro 27. Comando para gerar a documentação do sistema estrutural 49
Quadro 28. Método comentado do sistema MVC. 50
Quadro 29. Comando para gerar a documentação do sistema MVC 50

Lista de Tabelas

Tabela 1. Tags do PHPDocumentor 39
Tabela 2. Opções atribuídas aos resultados de cada sistema 52
Tabela 3. Resultado da implementação de novos componentes 52
Tabela 4. Resultado das mudanças de regras de negócio 53
Tabela 5. Resultado da geração da documentação 53

Resumo

Esta pesquisa se trata de um estudo sobre padrões de desenvolvimento em ambiente web utilizando a linguagem de script PHP, motivada pelo fato do PHP ser multiparadigma, suportando o paradigma estrutural e orientado a objetos, a linguagem abrange inúmeras formas para o desenvolvimento de software, resultando em problemas de consistência. Ao mesmo tempo que os desenvolvedores podem desfrutar da liberdade oferecida, é possível que técnicas e boas práticas de programação sejam aplicadas de forma incorreta, podendo obter uma qualidade insatisfatória do software. Este estudo abrange conceitos de padrões de projeto, paradigma de orientação a objetos e aborda os problemas estruturais do PHP, com o objetivo de auxiliar os desenvolvedores web a projetarem sistemas inteligentes que possuam longa manutenibilidade e possam suprir os requisitos de forma satisfatória com menor perda de tempo e recursos financeiros. Para a metodologia foram desenvolvidos dois sistemas, com o objetivo de realizar simulações que implementam novos componentes, mudam regras de negócio e geram a documentação. No sistema estrutural foi realizado a refatoração de código, transformando-o em orientado a objetos e arquitetado com MVC, com a análise dos problemas encontrados foram aplicados padrões, como Front Controller, Injeção de Dependência, Strategy e PSR, com o intuito de desenvolver um sistema reutilizável que atendesse de forma satisfatória as simulações. Pode-se concluir que é possível otimizar o tempo de desenvolvimento e manutenção em sistemas PHP, reutilizar uma grande parte dos componentes genéricos desenvolvidos e melhorar a qualidade do sistema, o ponto fundamental da engenharia de software.

Palavras-chave: PHP. Orientação a Objetos. Padrões de Projeto. Manutenibilidade. Engenharia de Software.

Abstract

This research is a study of web development standards using the PHP script language, motivated by the fact that it is multiparadigm language, supporting the structural and object-oriented paradigm. The language covers numerous ways of software development, resulting in problems of consistency. While developers can enjoy the offered freedom, its possible that good programming practices and tecniques be applied in wrong way, leading to unsatisfed software quality. This study covers concepts of design patterns, object-oriented paradigm and discussions about PHP structure problems, with the goal of helping web developers to design intelligent systems that have long maintainability and can satisfactorily achieve the requirements with less loss of time and financial resources. For the methodology, two systems were developed with the objective of performing simulations of implementing new components, change in business rules and documentation generation. In the structural system, the code refactoring was performed, transforming it into object oriented and MVC architected. With the analysis of the problems found, were applied some standards, such as Front Controller, Dependency Injection, Strategy and PSR, with the aim of developing a reusable system that met the simulations satisfactorily. It can be concluded that it is possible to optimize development and maintenance time in PHP systems, reuse a large part of the generic components developed and improve the quality of the system, the fundamental point of software engineering.

Keywords: PHP. Object-Oriented. Design Patterns. Maintainability. Software Engineering.

Sumário

LISTA DE FIGURAS V
LISTA DE QUADROS VI
LISTA DE TABELAS VII
RESUMO VIII
ABSTRACT IX
1 INTRODUÇÃO 12
2 CONCEITOS DE PROJETO 15
2.1 Projeto na engenharia de software 15
2.2 Arquitetura de software 16
2.3 Manutenção e o processo de refatoração 17
2.4 Padrões de projeto 18
2.4.1 Model-View-Controller 19
2.4.2 Strategy 21
2.4.3 Façade 22
2.4.4 Injeção de dependência 23
2.4.5 Front controller 24
3 LINGUAGEM PHP 25
3.1 História e evolução 25
3.2 Orientação a objetos 26
3.2.1 Objetos e classes 26
3.2.2 Encapsulamento 27
3.2.3 Métodos 28
3.2.4 Herança 29
3.2.5 Abstração 30
3.2.6 Interfaces 31
3.2.7 Polimorfismo 32
3.3 Padrões para codificação 34
3.4 Documentação para sistemas PHP 37
3.5 Frameworks 39
3.6 Principais problemas da linguagem PHP 40
3.6.1 Combinação entre códigos HTML e PHP 40

3.6.2 Programação estrutural X Orientada a objetos 40
4 MATERIAIS E MÉTODO 42
4.1 Refatoração de código estrutural para orientado a objetos com MVC 42
4.2 Implementando novos componentes 47
4.3 Modificando as regras de negócio 48
4.4 Gerando a documentação 48
5 RESULTADOS 52
5.1 Resultado da implementação de novos componentes 52
5.2 Resultado da modificação das regras de negócio 53
5.3 Resultado da geração da documentação 53
CONCLUSÃO 54
REFERÊNCIAS BIBLIOGRÁFICAS 55
APÊNDICE A 58

1 INTRODUÇÃO

Os softwares de computadores são produtos únicos capazes de resolver problemas de diversas áreas, se transformaram em uma tecnologia indispensável para negócios, ciência e engenharia (PRESSMAN, 2011). Há cinquenta anos atrás, os softwares eram entregues fora do prazo, com defeitos e não atendiam as necessidades do cliente, sendo assim a cada ciclo de manutenção que era realizado, tempo e recursos financeiros eram esgotados comparando com uma criação de um novo software. Portanto novas tecnologias foram criadas e aprimoradas para auxiliar no desenvolvimento, a engenharia de software é uma tentativa de resolver esses problemas, ela tem como objetivo criar softwares de alta qualidade (SCHACH, 2011).

Para desenvolver um software de qualidade é necessário projetar, o que é um processo importante da engenharia de software, onde reúne um conjunto de conceitos e práticas que permitem modelar os requisitos e a arquitetura de um sistema antes da implementação do código. Sem um projeto de software é possível se desenvolver um sistema com muitas falhas, sendo impossível de implementar novos recursos, tendo maior dificuldade em realizar testes e de ser reutilizado futuramente (PRESSMAN, 2011).

Um princípio clássico é o projeto modular, onde é possível aplicar a modularidade em um sistema, ou seja, projetar módulos coesos e não acoplados. A coesão é um conceito que define o quanto as responsabilidades de um módulo estão distribuídas. O acoplamento é um conceito que define o grau de dependência entre os módulos. Existe um equilíbrio sobre a coesão e acoplamento, o mais desejável é uma alta coesão e um baixo acoplamento, levando em consideração que é impossível desenvolver um sistema cem por cento coeso e não acoplado, há casos onde será necessário acoplar algum módulo ou adicionar alguma função para satisfazer um problema específico do projeto, portanto esses conceitos apresentam várias vantagens como a facilidade de entendimento do projeto, simplificação da manutenibilidade, evolução do sistema e a reutilização dos módulos (LARMAN, 2007).

A arquitetura de software permite estruturar a base do sistema, organizar seus módulos e como eles se interagem, é uma junção entre o projeto e os requisitos extraídos das necessidades do cliente. A arquitetura de software é importante para garantir a funcionalidade e a robustez, sendo possível a capacidade de manutenibilidade e flexibilidade de um sistema (SOMMERVILLE, 2011).

O desenvolvimento de software requerer recursos financeiros para ser mantido e criado. Há uma grande necessidade de encontrar saídas para reduzir os custos, no caso os sistemas que são desenvolvidos seguindo principalmente a programação estrutural, tendem aumentar os custos financeiros com a manutenção. É possível resolver este problema utilizando a refatoração de código, uma importante técnica para melhorar a produção de software, que auxilia nas etapas de testes, produtividade e simplicidade no processo de desenvolvimento, sendo possível a adição de padrões de projeto para a extensão de novas funcionalidades. (TRUCCHIA; ROMEI, 2010).

Em um projeto de software com qualidade é recomendado utilizar os padrões de projeto, que são constituídos por um problema genérico com um conjunto de classes e são analisadas para compor um projeto específico (SCHACH, 2011). Essas soluções são utilizadas em larga escala em projetos orientados a objetos que permitem descrever o relacionamento entre as entidades, podendo resolver problemas no desenvolvimento (DALL'OGLIO, 2015). Os padrões de projeto não são a forma mais rápida e simples para resolver problemas, porém são a chave para desenvolver sistemas robustos e reutilizáveis que podem suportar mudanças futuras de requisitos (GAMMA et al., 2007). Esses padrões foram descritos primeiramente no livro Design Patterns em 1995 por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, também conhecidos como GoF (Gang of Four – Gangue dos Quatro), este livro aborda a aplicabilidade, estrutura e implementação de cada padrão de projeto (ZANDSTRA, 2013).

Dentro dos padrões de projeto existem os padrões arquiteturais que são testados e aprovados em diversas situações, sendo importantes para desenvolver uma arquitetura robusta. Um padrão de arquitetura deve ser analisado e escolhido para atender as necessidades do projeto a ser desenvolvido, neste contexto foi estudado o padrão MVC (Model-View-Controller – Modelo Visão Controlador), onde seu objetivo inclui separar as camadas da arquitetura do sistema, sendo assim, favorecendo uma baixa dependência entre objetos e um ótimo gerenciamento de responsabilidades. A arquitetura MVC é implementada na maioria dos projetos para web, sendo possível organizar as funcionalidades e separar a lógica do sistema com a interface do usuário (SOMMERVILLE, 2011).

A linguagem de script Hypertext Preprocessor, conhecida pela sigla PHP, é utilizada em larga escala no desenvolvimento web, segundo uma pesquisa realizada pela W3Techs (2016), a linguagem está presente em 82% dos websites. Sua estrutura comporta o paradigma de

programação estrutural e a partir da versão 3.0 passou a suportar o paradigma de orientação a objetos, uma solução que permite reduzir o nível de complexidade da lógica, a reutilização e manutenibilidade de um sistema (SCHACH, 2011).

A programação estrutural foi uma das causas dos problemas de software que ocorreram antes da engenharia de software ser reconhecida, essa abordagem começou a enfrentar problemas em sistemas maiores, onde a maioria do tempo era gasto com manutenção do sistema após ser entregue ao cliente. Foi onde o surgiu o paradigma de orientação a objetos em meados de 1980, uma nova técnica para desenvolver sistemas mais legíveis e reutilizáveis (SCHACH, 2011).

Como a linguagem PHP permite várias formas de desenvolvimento, consequentemente muitos sistemas poderão apresentar problemas ao realizar uma manutenção, novos recursos para negócios serão difíceis de serem implementados e técnicas de programação reconhecidas podem ser aplicadas de forma incorreta (DALL'OGLIO, 2015). A razão para ocorrer estes problemas é a liberdade que a linguagem oferece para o desenvolvimento estrutural, como iniciar um sistema por sua implementação e ignorar as fases de projeto, o hábito de misturar códigos em HTML (HyperText Markup Language – Linguagem de Marcação de Hipertexto) e PHP, inclusive os desenvolvedores precisam adotar alguns padrões individuais para controlar a complexidade do código (BAJGORIC, 2009).

Analisando os problemas que a linguagem PHP utilizando o paradigma estrutural pode proporcionar ao desenvolvimento web, é possível elaborar soluções orientada a objetos padronizadas, para isso foi desenvolvido dois sistemas de gerenciador de biblioteca para realizar simulações de implementação de novos componentes, mudanças de regras de negócio e geração de documentação. Foi realizado a refatoração do sistema desenvolvido sob o paradigma estrutural com nenhum tipo de padronização, com isso foi aplicado o paradigma de orientação a objetos, a arquitetura MVC para melhorar a organização da estrutura do sistema e padrões de projeto, que foram capazes de resolver os problemas encontrados.

Com as simulações realizadas, foi possível obter resultados de que o sistema arquitetado com MVC otimiza o tempo de desenvolvimento e manutenção, onde uma grande parte dos componentes podem ser reutilizados em sistemas diferentes, tendo como finalidade melhorar a qualidade dos sistemas em PHP.

2 CONCEITOS DE PROJETO

Este capítulo apresenta os princípios de projeto e arquitetura de software, introduzindo a importância de utilizar padrões no desenvolvimento, são abordados os padrões Model-View-Controller, Strategy, Façade, Injeção de Dependência e Front Controller.

2.1 Projeto na engenharia de software

No contexto da engenharia de software, um projeto permite modelar um sistema e ser analisado em termos de qualidade antes do código ser elaborado, segundo Larman (2007) o projeto enfatiza uma solução conceitual, podendo ser um software ou hardware que satisfaça os requisitos e não sua implementação. Diante disso, Pressman (2011, p. 207) comenta sobre projetos em seu livro de engenharia de software.

> O projeto de software reside no núcleo técnico da engenharia de software e é aplicado independentemente do modelo de processos de software utilizado. Iniciando assim que os requisitos de software tiverem sido analisados e modelados, o projeto de software é a última ação da engenharia de software da atividade de modelagem e prepara a cena para a construção (geração de código e testes) (PRESSMAN, 2011, p. 207).

Um projeto é a ampla visão do modelo de sistema, utilizando a notação diagramática padrão UML (Unified Modeling Language - Linguagem de Modelagem Unificada), permite apresentar uma planta do software para auxiliar nas decisões dos fluxos de dados de sistema orientado a objetos (LARMAN, 2007), além disso com um refinamento mais aprofundado sobre o problema abordado, o modelo evolui para uma visão mais detalhada contribuindo para a especificação de novas funcionalidades, integrando uma arquitetura para definição estrutural dos mecanismos do sistema, padrões para desenvolvimento com práticas comprovadas, o reconhecimento de separar as responsabilidades de cada parte do sistema, com o objetivo de auxiliar o entendimento, testes e manutenção, encapsulamento para proteger as informações e reduzir a disseminação de erros, atribuir a independência entre os módulos quando for necessário, e a importância das classes orientadas a objetos, o paradigma mais utilizado na engenharia de software moderna. A ausência de um projeto de software aumenta os riscos de desenvolver um sistema instável, provocando falhas ao receber mudanças e dificulta os testes necessários para entregar um produto final com qualidade (PRESSMAN, 2011).

De acordo com Pressman (2011, p. 354), projetos para sistemas web contém em sua essência os mesmos conceitos de um projeto de software, porém para os engenheiros web o desafio é o

compromisso com a segurança das informações em bancos de dados, a simplicidade da aparência que é composto pelo design da interface do usuário e a consistência do mecanismo de navegação entre as páginas web do sistema.

> A qualidade de uma WebApp – Definida em termos de usabilidade, funcionalidade, confiabilidade, eficiência, facilidade de manutenção, segurança, escalabilidade e tempo para colocação no mercado – é introduzida durante o projeto. Para alcançar tais atributos de qualidade, um bom projeto de WebApp deve apresentar as seguintes características: simplicidade, consistência, identidade, robustez, navegabilidade e apelo visual (PRESSMAN, 2011, p. 354).

2.2 Arquitetura de software

Em seu livro, Pressman (2011) cita os autores Shaw e Garlan onde abordam a arquitetura de software como um conjunto de módulos de um sistema, onde desde que o primeiro sistema foi dividido em módulos ou partes, passaram a conter uma arquitetura de software.

O projeto de arquitetura é o alicerce de um sistema, no livro Software Architecture in Practice, os autores Bass, Clements e Kazman (2013, p. 21) descrevem: “A arquitetura de um software é a estrutura do sistema, que compreendem elementos de software, as propriedades externamente visíveis de tais elementos, e as relações entre eles”, ou seja, o foco está na organização estrutural de módulos genéricos e seus processos para serem reaproveitados futuramente. A arquitetura em camadas possibilita a separação de responsabilidades e independência entre módulos, isso é fundamental para localizar alterações na estrutura. Como demonstrado na Figura 1, uma camada fornece serviços para outras camadas, os níveis mais baixos são representados por funcionalidades genéricas utilizadas por todo sistema (SOMMERVILLE, 2011).

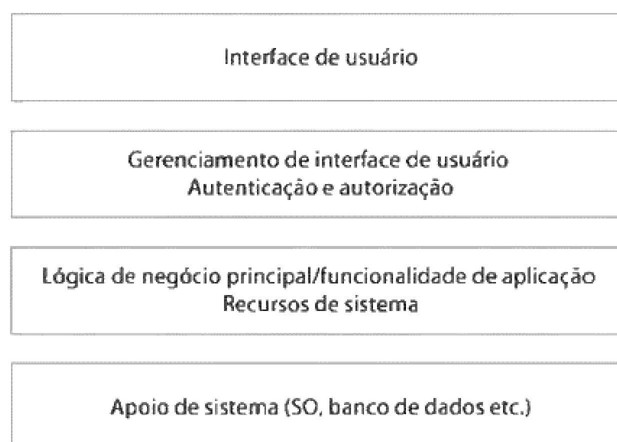

Figura 1. Uma arquitetura genérica em camadas
Fonte: SOMMERVILLE, 2011.

2.3 Manutenção e o processo de refatoração

A manutenção é um procedimento de mudanças que o software recebe depois de ser liberado para o cliente. Este procedimento é caracterizado por melhorias, correções de erros e expansão de requisitos. Segundo Sommerville (2011), pesquisas realizadas apontam que dois terços de um orçamento de TI são destinados a manutenção, e um terço para o desenvolvimento. Na Figura 2, a pesquisa também aponta que no processo de manutenção gasta-se mais na implementação de novos recursos do que na correção de falhas, portanto dependendo do domínio da aplicação os custos podem variar, como por exemplo, a manutenção para um sistema corporativo tem em média o mesmo custo de seu desenvolvimento, em comparação com um sistema embutido de tempo real, o custo da manutenção pode aumentar até quatro vezes.

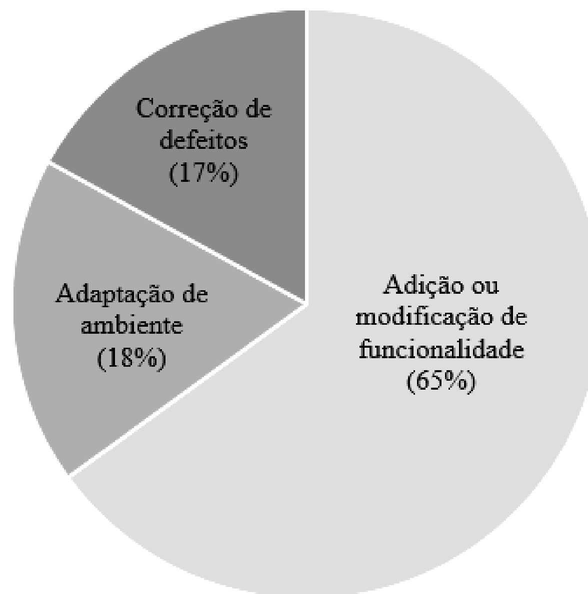

Figura 2. Distribuição de esforço de manutenção
Fonte: SOMMERVILLE, 2011.

Adicionar novas funcionalidades em um sistema já entregue é caro e os riscos para gerar um código com falhas aumentam consideravelmente (TRUCCHIA; ROMEI, 2010), neste caso é importante investir nas fases de projeto e implementação, sendo possível visualizar as necessidades e adicionar os novos recursos antes do sistema ser entregue, para isso utiliza-se as técnicas oferecidas pela engenharia de software, como analise de requisitos, o uso do paradigma de orientação a objetos e o processo de refatoração que contribuem para a redução de custos na manutenção (SOMMERVILLE, 2011).

A refatoração é um processo que transforma e preserva o comportamento da estrutura, ou seja, é uma alteração feita no sistema para transformá-lo em mais legível e mais barato de ser modificado sem alterar seu comportamento. Conforme o autor Kerievsky (2008), todo o processo de refatoração inclui a remoção de código duplicado, simplificação da lógica envolvida e a possibilidade de adicionar novos recursos. Este processo é realizado durante o desenvolvimento, diferente da reengenharia de software, que ocorre em situações onde um sistema já foi mantido por algum tempo e teve aumentos em sua manutenção (SOMMERVILLE, 2011).

2.4 Padrões de projeto

Para introduzir a definição de padrões de projeto, é importante citar o autor Christopher Alexander, um acadêmico no qual teve seu trabalho divulgado em 1977 que foi fortemente influenciado nos padrões originais de orientação a objetos, ele afirma que cada padrão descreve um problema que ocorre eventualmente em nosso ambiente e em seguida, descreve o núcleo da solução para este problema, onde é possível utilizar a solução inúmeras vezes, mas nunca utiliza-lo da mesma forma duas vezes. Portanto essa ideia de padrão de projeto foi propagado no meio da tecnologia da informação pelo grupo GoF, que teve uma importante contribuição para soluções reutilizáveis de software orientado a objetos contando com seus 23 padrões (ZANDSTRA, 2013). O objetivo é catalogar a experiência em projetos orientados a objetos em aspecto de padrões, cada padrão nomeia, explica e avalia uma solução. A utilização de padrões facilita o reuso de projetos e arquiteturas bem definidas, a demonstração de técnicas que são aprovadas e testadas auxiliam os desenvolvedores em novos projetos, são aperfeiçoadas a manutenibilidade e a documentação do sistema ao proporcionar as interações entre os objetos (GAMMA et al., 2007). Cada padrão é independente da linguagem de programação, contudo é necessário que a linguagem suporte os conceitos do paradigma de orientação a objetos para serem implementados de forma satisfatória (DALL'OGLIO, 2015).

Com o sucesso dos padrões de projeto GoF, o autor e pesquisador Fowler (2006) publicou seu livro intitulado de Padrões de Arquitetura de Aplicações Corporativas, com o intuito de desenvolver padrões direcionados a projetos corporativos, tendo como exemplo, um sistema financeiro que requer uma exibição, manipulação e armazenamento de grandes quantidades de informações complexas de uma forma inteligente. Com esta necessidade foram desenvolvidos

diversos padrões, nesta pesquisa foram utilizados o Front Controller e Injeção de Dependência, padrões para web que acompanham o Model-View-Controller.

Os padrões de arquitetura são compostos por uma descrição abstrata, estilizada, de boas práticas testadas e recomendadas que foram implementadas em diversos sistemas, em sua documentação deve existir informações de quando é necessário utilizar esse padrão, apresentar seus pontos positivos e negativos (SOMMERVILLE, 2011). Em conformidade com o autor Fowler (2006), o padrão arquitetural Model-View-Controller é um dos padrões mais citados, foi descrito como um framework desenvolvimento por Trygve Reenskaug para a linguagem Smalltalk no final da década de 1970, com isso influenciou diversos projetos orientado a objetos por suprir as necessidades em um projeto de arquitetura de software.

2.4.1 Model-View-Controller

O Model-View-Controller é constituído pelas suas três camadas descritas na Figura 3, a primeira é a camada modelo, responsável pela organização das regras de negócio, a segunda é a visão, responsável pela interface do usuário com intuito de renderizar as informações e a terceira é a camada do controlador, responsável pelo controle do fluxo de informações e requisições das outras camadas. Os padrões de projeto mais explícitos que estão presentes no MVC são, Strategy, Observer e Composite (GAMMA et al., 2007). Sua estrutura pode variar dependendo da plataforma em que será implementado ou como a arquitetura foi planejada. Segundo o autor Sommerville (2011, p. 110), o modelo MVC desacopla os módulos de um sistema favorecendo a reusabilidade da arquitetura.

> Quando uma camada é desenvolvida, alguns dos serviços prestados por ela podem ser disponibilizados para o usuários. A arquitetura também é mutável e portável. Enquanto sua interface for inalterada, uma camada pode ser substituída por uma outra equivalente. Além disso, quando a camada de interfaces muda ou tem novos recursos adicionados, apenas a camada adjacente é afetada. Como sistemas em camadas localizam dependências das máquinas em camadas internas, isso facilita o fornecimento de implementações multiplataforma de um sistema de aplicação (SOMMERVILLE, 2011, p. 110).

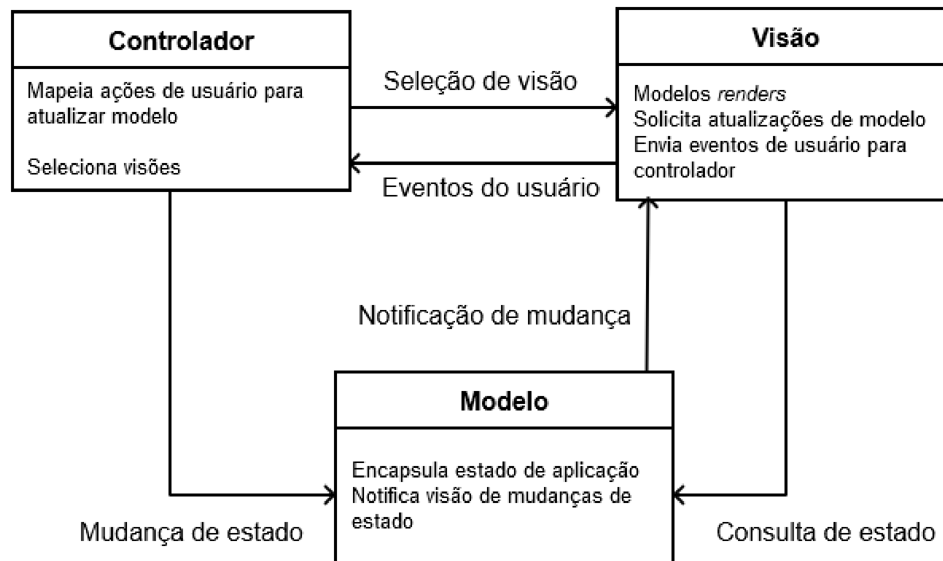

Figura 3. A organização do MVC
Fonte: SOMMERVILLE, 2011.

O MVC também pode ser utilizado no cenário web, sendo o principal padrão para o gerenciamento das tarefas que envolvem as aplicações, facilitando o trabalho em equipe, permitindo que os desenvolvedores trabalhem na mesma aplicação sem interferir um ao outro (SOMMERVILLE, 2011). O motivo para o padrão MVC ser popular em aplicações web é a organização estrutural que a separação de responsabilidades oferece para os engenheiros e sua equipe, é um conceito de projeto que é aplicado no contexto da separação das camadas do MVC, um problema complexo pode ser dividido em várias partes menores sendo mais fáceis de serem resolvidos de forma independente, gerando pequenos módulos para atender requisitos específicos do sistema e manter um código reutilizável. Em seu livro, Pressman (2011, p. 349) cita o autor Jacyntho e seus colegas, e apresenta a justificativa de utilizar o padrão MVC para sistemas web.

> Os autores sugerem uma arquitetura de projeto em três camadas que desassocia a interface da navegação e do comportamento da aplicação. Eles argumentam que manter a interface, a aplicação e a navegação separadas simplifica a implementação e aumenta a reutilização (PRESSMAN, 2011, p. 349).

Um modelo genérico de MVC para sistemas web necessita de outras camadas para gerenciar funções específicas de comunicação. A camada rotas é um padrão para gerenciar diversas URL’s (Uniform Resource Locator – Localizador Padrão de Recursos) pré-definidas que são solicitadas pelo navegador, o sistema tenta buscar uma rota correspondente, e se for encontrada,

é chamada uma ação do controlador associado a este percurso (COLEMAN, 2016). A Figura 4 demonstra um modelo de MVC genérico para sistemas web com a camada rotas inclusa.

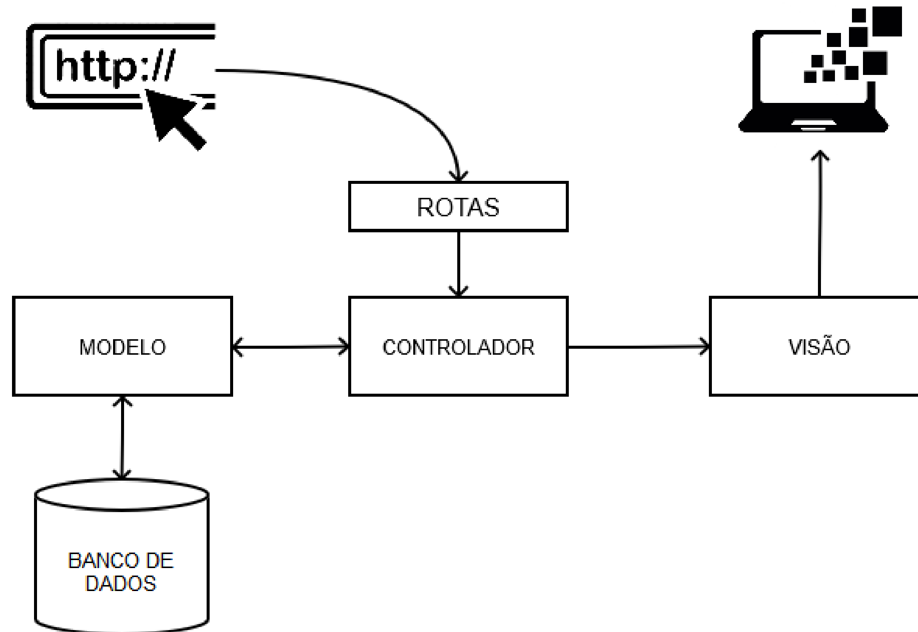

Figura 4. O fluxo MVC com as rotas inclusas
Fonte: COLEMAN, 2016.

2.4.2 Strategy

Strategy é um padrão de projeto GoF do tipo comportamental de objetos, no qual são responsáveis pela atribuição de responsabilidades e comunicação entre os objetos do sistema, no caso do padrão Strategy ou Estratégia, ele é responsável pelo gerenciamento do comportamento dos objetos (GAMMA et al., 2007). O problema abordado é como projetar algoritmos ou regras que estão relacionadas e como projetar para que possam sofrer mudanças, no entanto de acordo com a sugestão do autor Larman (2007), é importante definir regras ou estratégias em uma classe que implementa uma interface em comum, ou seja, criar interfaces com funcionalidades genéricas que serão utilizadas por classes diferentes, estes conceitos de orientação a objetos são esclarecidos no Capítulo 3 desta pesquisa.

Demonstrando na Figura 5, um diagrama de classes da UML, é possível visualizar o padrão Estratégia utilizando uma interface com duas classes que estão herdando seus métodos polimórficos, podendo receber valores diferentes sem interferir em outras classes.

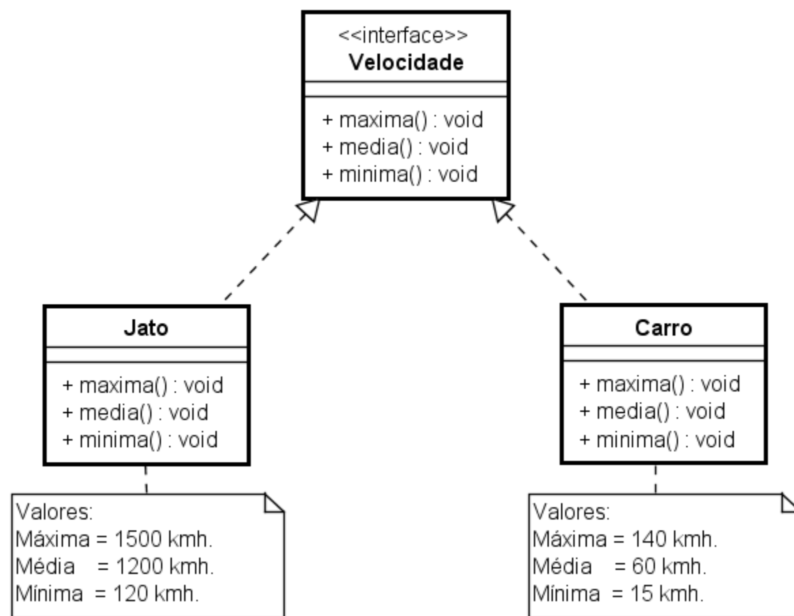

Figura 5. Exemplo de padrão Estratégia
Fonte: Autoria própria, 2016.

2.4.3 Façade

Façade é um padrão de projeto GoF do tipo estrutural de objetos, contém a responsabilidade de descrever maneiras de criar objetos para conquistar novas funcionalidades, ao contrário dos padrões estruturais de classes, que utilizam a herança para estruturar implementações ou interfaces. Um objeto Façade ou Fachada, demonstra como uma classe pode representar todo o módulo, seu objetivo é fornecer um objeto de alto nível para um conjunto não uniforme de implementações ou interfaces que pertencem ao módulo, tornando-se a separação de responsabilidades algo desejável (GAMMA et al., 2007). Aplica-se este padrão amplamente utilizado quando é necessário ocultar um subsistema por trás de um objeto, como separar as camadas de domínio e tela do usuário (LARMAN, 2007). Inclusive é utilizado em situações onde ocorre a necessidade de blindar partes do sistema de bibliotecas de terceiros e também de códigos legados (DALL'OGLIO, 2015).

Na Figura 6 é demonstrado um exemplo utilizando um diagrama de classes da UML, as classes do primeiro módulo são dependentes da fachada para acessar os dados do segundo módulo, esta fachada inclui uma relação de associação unidirecional com a interface regras, onde a fachada

tem conhecimento sobre as regras, mas as regras não tem conhecimento sobre a associação realizada, isso desacopla os objetos tornando o sistema reutilizável (BELL, 2004).

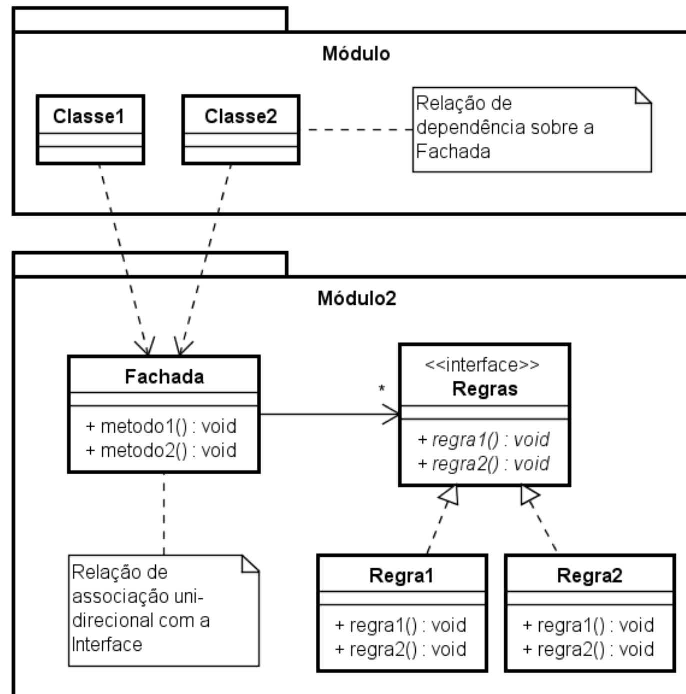

Figura 6. Exemplo de padrão Fachada
Fonte: Autoria própria, 2016.

2.4.4 Injeção de dependência

A injeção de dependência é um padrão de projeto desenvolvido pelos pesquisadores Robert C. Martin e Martin Fowler (DALL'OGLIO, 2015), que visa resolver os problemas de acoplamento de classes de um sistema orientado a objetos, é amplamente utilizado na arquitetura MVC para resolver problemas de dependências (PRETTYMAN, 2016). A dependência ocorre quando uma classe precisa utilizar recursos de outra, para isso é criado uma instância da classe que possui os métodos e atributos de interesse. Aplicando a injeção de dependência, a classe não irá mais realizar a instância do objeto no qual depende, porém passará a ser recebida através de parâmetros como uma "injeção", onde não precisará mais importar-se com a responsabilidade de instanciação do objeto, que deverá ser realizada uma camada acima.

Em um exemplo prático, uma classe que precisa executar consultas em um banco de dados, depende explicitamente da conexão, utilizando o modelo tradicional sem a injeção de dependência, a classe iria instanciar a conexão e realizar as consultas, supondo que seja preciso alterar a conexão ou utilizar uma secundária, seria mais complexo de se realizar, pois a classe

está presa à sua implementação, sendo necessário mudanças no código. Com a injeção de dependência, a classe realizará a instância do objeto para efetuar as consultas, geralmente um controlador, a conexão é instanciada e passará por parâmetro para o objeto encarregado em gerenciar as consultas, ou seja, a classe que irá realizar as consultas no banco de dados não é mais responsável em instanciar o banco de dados, pois este sempre estará disponível, através de uma injeção de dependência, caso seja preciso utilizar conexões secundárias ou realizar modificações, nenhuma alteração será feita, apenas no objeto instanciado, evitando acoplamento direto entre ambas (LOCKHART, 2015). Este padrão também contribui para um desenvolvimento voltado para testes integrais de módulos do sistema, reduzindo custos de manutenção pós entrega (RODRIGUE, 2012).

2.4.5 Front controller

O padrão de projeto para apresentação web Front Controller desenvolvido por Fowler (2006, p. 328), é especificado como um controlador que possui a capacidade de tratar todas as solicitações de um sistema, normalmente é representado como uma única classe responsável pelas requisições realizadas no sistema, despachando-as para as classes correspondentes.

> Em um site web complexo, há muitas coisas semelhantes que você precisa fazer ao tratar uma solicitação. Estas coisas incluem segurança, internacionalização e fornecimento de apresentações particulares para determinados usuários. Se o comportamento do controlador de entrada for espalhado por vários objetos, muito deste comportamento pode acabar duplicado. Além disso, é difícil alterar comportamento em tempo de execução. O controlador frontal consolida todo o tratamento de solicitações canalizando-as através de um único objeto manipulador. O manipulador então despacha para objetos do tipo comando que possuem comportamento específico relacionado a solicitação (FOWLER, 2006, p. 328).

O Front Controller normalmente é utilizado com a arquitetura MVC e aplicações complexas, além disso pode conter várias abordagens diferentes de acordo como foi projetado o sistema, aplicando este padrão em um software web, poderá apresentar um conjunto de páginas independentes beneficiando um desenvolvimento modular, inclusive tendo um ponto central de acesso que gerencia as requisições das páginas que devem ser exibidas, sendo possível implementar novas funcionalidades (DALL’OGLIO, 2015).

3 LINGUAGEM PHP

Este capítulo introduz a história e evolução da linguagem PHP, foram pesquisados os benefícios do paradigma de orientação a objetos, incluindo seus conceitos como objetos e classes, encapsulamento, métodos, herança, abstração, interfaces e polimorfismo, com implementações de código seguindo os padrões PSRs, um conjunto de normas com o objetivo de padronizar a codificação.

3.1 História e evolução

A linguagem PHP inicialmente significava Personal Home Page Tools, foi criada pelo canadiano-dinamarquês Rasmus Lerdorf no final de 1994 com o propósito de gerenciar as visitas em seu currículo na internet. Ferramentas que auxiliavam o desenvolvimento web não existiam naquela época, Lerdorf começou a disponibilizar a linguagem para outros desenvolvedores que necessitavam de uma solução para web, com isso o desenvolvimento foi incentivado e em 1997 o PHP já suportava a criação de formulários e comunicação com banco de dados.

Ao longo das versões novos desenvolvedores se aliaram ao desenvolvimento do PHP, Zeev Suraski, Andi Gutmans e Lerdorf começaram a discutir quais melhorias que a linguagem poderia receber, em 1998 foi lançada a versão 3.0 atualizando o nome da linguagem para Hypertext Preprocessor, com aprimoramento de performance para o usuário, conexão com vários bancos de dados, consistência da sintaxe e suporte ao paradigma de orientação a objetos. No mesmo ano Zeev e Andi começaram a reescrita do núcleo do PHP, com o objetivo de melhorar a modularidade e desempenho em aplicações mais complexas, o resultado levou o nome do novo núcleo de Zend Engine.

Em 2000 foi lançada a versão 4.0 já possuindo o novo núcleo, trazendo melhorias a sessões, suporte a vários tipos de servidores e permitindo ser utilizado como linguagem de shell script, mas ainda a linguagem necessitava de um suporte robusto para o paradigma de orientação a objetos, isso ocorreu em 2004 com o lançamento da versão 5.0, que possibilitou o desenvolvimento de projetos corporativos utilizando o paradigma e o padrão Model-View-Controller (DALL’OGLIO, 2015).

3.2 Orientação a objetos

Antes da abordagem orientada a objetos ser popularizada, nos períodos de 1975 e 1985, foram feitas grandes soluções com o desenvolvimento estruturado, estas eram compostas por análise estrutural, análise de fluxo de dados, programação e testes estruturados, portanto com o passar do tempo novas tecnologias e necessidades corporativas surgiram e esse tipo de abordagem clássica começou a enfrentar problemas no desenvolvimento de software. Por muitas vezes as técnicas estruturais eram incapazes de lidar com sistemas de grande porte que de fato geravam códigos desorganizados, ou seja, as técnicas foram criadas para suportar sistemas menores com uma estimativa de 5.000 linhas de código ou de médio porte com 50.000 linhas. Outro ponto fundamental do problema era o gasto de recursos financeiros e o tempo com manutenção do sistema após ser entregue, no qual chegava a ser cerca de 70% (SCHACH, 2011).

O paradigma de programação orientada a objetos é uma abordagem para elaboração de sistemas, que envolve desde a modelagem do sistema até seu desenvolvimento utilizando objetos que se relacionam (DALL'OGLIO, 2015), é um modelo de objeto engloba princípios da abstração, encapsulamento, modularidade e herança (BOOCH, 1994), os conceitos começaram a ganhar notabilidade durante as décadas de 1980 e 1990 quando a programação estrutural ainda era dominante entre os desenvolvedores. De acordo com o autor Schach (2011, p. 20), o paradigma aplicado de forma correta incentiva a prática de reutilização de código, simplificando o desenvolvimento e a manutenibilidade de um sistema.

> O paradigma de orientação a objetos reduz o nível de complexidade de um produto de software e, portanto, simplifica tanto o desenvolvimento quanto a manutenção. O paradigma de orientação a objetos permite reutilização; pelo fato de os objetos serem entidades independentes, eles podem ser utilizados em novos produtos futuros. Essa reutilização de objetos reduz o tempo e o custo, tanto em termos de desenvolvimento quando de manutenção. (SCHACH, 2011, p. 20)

3.2.1 Objetos e classes

Na elaboração de um sistema completo aderindo a este paradigma, requer a criação de objetos que gerenciem cada funcionalidade específica do sistema, no processo de construção de um objeto é necessário expor sua estrutura contendo os atributos e comportamento. Esta estrutura denomina-se como classe. Os objetos são representados por variáveis que apontam para uma região da memória herdando atributos e métodos da classe (DALL'OGLIO, 2015), segundo Pressman (2011), por definição os objetos são instâncias de uma classe específica e herdam seus atributos e propriedades públicas.

Conforme o Quadro 1, a declaração de uma classe no PHP é feita com a palavra reservada class, contendo em sua estrutura os modificadores de acesso denominados como public, protected ou private para definir sua visibilidade dentro do sistema.

Quadro 1. Exemplo de classe e instanciação do objeto

```
<?php
namespace Exemplos;

class MinhaClasse
{
    public    $atributo1;
    private   $atributo2;
    protected $atributo3;

    public function meuMetodo()
    {
        echo 'meuMetodo() foi imprimido';
    }
}

// Instanciação do objeto
$obj = new MinhaClasse;
```

Fonte: Autoria própria, 2016.

3.2.2 Encapsulamento

O encapsulamento é um dos recursos significativos do paradigma, é possível isolar atributos importantes que não devem ser acessados de forma direta por outas entidades. Em seu livro, Dall'Oglio (2015, p. 121) aborda o encapsulamento como um recurso que favorece proteção interna dos atributos ou métodos de um objeto.

> Um dos recursos mais interessantes na orientação a objetos é o encapsulamento, um mecanismo que provê proteção de acesso aos membros internos de um objeto. Lembre-se de que uma classe tem responsabilidade sobre os atributos que contém. Dessa forma, existem certas propriedades de uma classe que devem ser tratadas exclusivamente por métodos dela mesma, que são implementações projetadas para manipular essas propriedades da forma correta. As propriedades não devem ser acessadas diretamente de fora do escopo de uma classe, pois dessa forma a classe não fornece mais garantias sobre os atributos que contém, perdendo, assim, a responsabilidade sobre eles. (DALL'OGLIO, 2015, p. 121)

Para definir o encapsulamento dos dados é necessário estabelecer a visibilidade dos atributos e dos métodos do objeto. A visibilidade pública permite o acesso livre de seus atributos e métodos, podendo ser acessadas pela própria classe e outros objetos do sistema. A visibilidade privada permite o acesso somente em sua classe, outros objetos não poderão ter visibilidade desses atributo ou métodos. A visibilidade protegida permite o acesso somente em sua classe e em objetos que foram herdados.

3.2.3 Métodos

Segundo o autor Dall'Oglio (2015), um método é uma função que faz parte de uma classe e representa um comportamento que ela necessita expor para os demais objetos, ou seja, é o processo onde é adicionado um comportamento à classe oferecendo funcionalidades ao ambiente externo, como calcular valores, validação de campos ou realizar uma consulta com o banco de dados. No Quadro 2 é demonstrado a criação de métodos, para isso é necessário definir a visibilidade do método diante dos outros objetos, em seguida utilizar a palavra reservada function.

Quadro 2. Exemplo de métodos

```
<?php
namespace Exemplos;

class Produto
{
    private $descricao;

    public function setDescricao($descricao)
    {
        if(is_string($descricao)){
            $this->descricao = $descricao;
        }
    }
    public function getDescricao()
    {
        return $this->descricao;
    }
}
```

Fonte: Autoria própria, 2016.

É possível criar objetos com atributos pré-definidos a partir do método construtor do PHP, onde este é chamado automaticamente logo após a criação de um objeto (ZANDSTRA, 2013) utilizando a palavra reservada new. O método nomeia-se como __construct(), a sua estrutura pode conter instruções ou até mesmo preenchimento de atributos que podem ser validados antes de ser atribuído ao objeto.

Outro método existente do PHP é o destrutor, denominado como __destruct() tem a função de retirar um objeto da memória, normalmente é chamado automaticamente quando o objeto está sendo destruído, como ao utilizar-se o método unset() ou quando o script é finalizado completamente onde todos os objetos são destruídos (DALL'OGLIO, 2015).

3.2.4 Herança

A herança é um elemento essencial para um sistema orientado a objetos, onde é possível definir um relacionamento entre as classes e compartilhar sua estrutura e comportamento com outras entidades do sistema. De acordo com Booch (1994) hereditariedade representa, assim, uma hierarquia de captações, em que uma subclasse herda uma ou mais superclasses.

O conceito de herança pode ser denominado como uma hierarquia de generalização e especialização. As superclasses são representações de abstrações generalizadas e as subclasses representações de abstrações especializadas em que os atributos e métodos da superclasse são adicionadas ou ocultadas (BOOCH, 1994). Segundo o autor Dall'Oglio (2015, p. 113), a herança beneficia a reutilização de código em um sistema.

> Esse recurso tem uma aplicabilidade muito grande, visto que é relativamente comum termos de criar novas funcionalidades em software. Utilizando a herança, em vez de criarmos uma estrutura totalmente nova (uma classe), podemos reaproveitar uma estrutura já existente que nos forneça uma base abstrata para o desenvolvimento, provendo recursos básicos e comuns (DALL'OGLIO, 2015, p. 113).

O Quadro 3 apresenta um exemplo de herança, o namespace "Exemplos" é constituído por uma classe pai, contendo em sua estrutura dois atributos protegidos e um método construtor.

Quadro 3. Exemplo de herança - Classe Pai

```php
<?php
namespace Exemplos;

class ClassePai
{
    protected $atributo1;
    protected $atributo2;

    public function __construct($atributo1, $atributo2)
    {
        $this->atributo1 = $atributo1;
        $this->atributo2 = $atributo2;
    }
}
```

Fonte: Autoria própria, 2016.

Como demonstrado no Quadro 4, a classe filha é constituída por um atributo privado e herdará a estrutura da classe pai, a herança é feita pela palavra reservada extends, o método construtor foi sobrescrito e adicionado o atributo $atributo3, este processo é denominado de override, dentro do mesmo foi chamado o método construtor da classe pai utilizando a palavra reservada parent.

Quadro 4. Exemplo de herança - Classe Filha

```
<?php
namespace Exemplos;

use Exemplos\ClassePai as ClassePai;

class ClasseFilha extends ClassePai
{
    private $atributo3;

    // Sobrescrita de método (override)
    public function __construct($atributo1, $atributo2, $atributo3)
    {
        // Chamando o método construtor da classe pai
        parent::__construct($atributo1, $atributo2);

        $this->atributo3 = $atributo3;
    }
}
```

Fonte: Autoria própria, 2016.

3.2.5 Abstração

A abstração é uma das formas mais importantes que os seres humanos conseguem lidar com a complexidade, ela surge a partir de um reconhecimento das características semelhantes entre o mundo real e objetos de um sistema. Uma abstração é realizada com uma visão de fora de um objeto, podendo assim separar o comportamento a partir da sua implementação (BOOCH, 1994). No desenvolvimento orientado a objetos é necessário separar o sistema em várias partes menores, focando-se nas funcionalidades mais importantes onde é possível construir partes do sistema que podem ser reutilizadas futuramente formando uma estrutura hierárquica (DALL'OGLIO, 2015).

No PHP orientado a objetos é possível construir classes abstratas que podem ser utilizadas como base para outras classes do sistema. O Quadro 5 apresenta uma estrutura onde estão presentes funcionalidades genéricas que outras classes filhas poderão utilizar. Uma classe abstrata não pode ser instanciada como objeto, somente suas classes filhas.

Quadro 5. Exemplo de classe abstrata

```
<?php
namespace Exemplos;

abstract class ClasseAbstrata
{
    public function helloWorld()
    {
        $this->imprimir('Hello World');
    }
}
```

Fonte: Autoria própria, 2016.

Métodos abstratos também são possíveis de serem criados dentro de uma classe abstrata, são definidos o nome e seus parâmetros, sua implementação será desenvolvida quando esse método abstrato for herdado na classe filha (DALL'OGLIO, 2015).

3.2.6 Interfaces

Interfaces é um conceito da orientação a objetos que auxilia o baixo acoplamento entre classes de um sistema. O conceito de acoplamento significa o grau de dependência que uma classe tem sobre as outras, quanto maior o acoplamento mais difícil será reutilizar essas funcionalidades, com isso é recomendado utilizar-se do baixo acoplamento, contudo é impossível desenvolver um sistema cem por cento desacoplado, de fato é necessário possuir os relacionamentos entre classes para o funcionamento do sistema. Uma interface tem o papel de representar uma fachada, sua estrutura é abstrata onde são criados métodos que serão implementados por outras classes de forma obrigatória (DALL'OGLIO, 2015). A criação de uma interface no PHP é demonstrada no Quadro 6, para isso é necessário utilizar a palavra reservada interface, e criar os métodos desejados.

Quadro 6. Exemplo de interface

```
<?php
namespace Exemplos;

interface MinhaInterface
{
    public function helloWorld();
    public function imprimir($texto);
}
```

Fonte: Autoria própria, 2016.

Para implementar uma interface em uma classe utilizamos a palavra reservada implements como demonstrado no Quadro 7, sendo obrigatório adicionar os mesmos métodos da interface, ou então será gerado um Fatal Error.

Quadro 7. Exemplo de implementação da interface

```php
<?php
namespace Exemplos;

use Exemplos\MinhaInterface as MinhaInterface;

class ImplementarInterface implements MinhaInterface
{
     //Métodos da interface
    public function helloWorld()
    {
        $this->imprimir('Hello World');
    }
    public function imprimir($texto)
    {
        echo $texto;
    }
}
```

Fonte: Autoria própria, 2016.

3.2.7 Polimorfismo

Polimorfismo é um conceito que permite classes herdadas de outras superclasses com os mesmo métodos, mas adquirindo comportamentos diferentes (DALL'OGLIO, 2015). O verdadeiro valor do polimorfismo está nas diferentes implementações de uma interface, onde objetos com uma interface em comum podem ser implementados para realizar funções diferentes, portanto é capaz de contribuir para sistemas de grande porte proporcionando um código mais inteligente (SANDERS, 2013). O autor Larman (2007, p. 428) explica qual a melhor situação para utilizar-se interfaces em um sistema.

> Polimorfismo implica na presença de superclasses abstratas ou interfaces na maioria das linguagens OO. Quando você deve considerar o uso de uma interface? A resposta geral é introduzir uma quando você deseja apoiar polimorfismo sem se comprometer com uma particular hierarquia de classes (LARMAN, 2007, p. 428).

Alguns padrões de projeto são constituídos pelo polimorfismo, dentre eles o padrão Estratégia inclui uma interface com diferentes objetos que integram as implementações, este conceito é discutido no Capítulo 2 em padrões de projeto. Exemplificando o padrão no PHP, pode-se incluir várias implementações da interface para diferentes ações em um banco de dados como as funções de incluir, alterar e excluir dados. Ao preservar uma interface em comum sobre os

objetos, novas funções específicas podem ser adicionadas sem influenciar no papel do objeto responsável pela manipulação dos dados de tabelas em SQL, com isso é possível melhorar a reutilização de código contribuindo para um baixo acoplamento (SANDERS, 2013).

Neste exemplo de polimorfismo demonstrado no Quadro 8, é criado uma interface chamada Velocidade, constituída por três métodos públicos, máxima(), media() e mínima() que serão implementadas em outras classes.

Quadro 8. Exemplo de polimorfismo utilizando interface

```
<?php
namespace Exemplos;

interface Velocidade
{
    public function maxima();
    public function media();
    public function minima();
}
```

Fonte: Autoria própria, 2016.

Logo em seguida nos Quadros 9 e 10 são criadas duas classes, uma chamada Jato e outra Carro, elas utilizarão a interface Velocidade implementando seus três métodos obrigatórios. Cada método retornará um velocidade específica referente a cada classe.

Quadro 9. Exemplo de polimorfismo - Classe Jato

```
<?php
namespace Exemplos;

use Exemplos\Velocidade as Velocidade;

class Jato implements Velocidade
{
    public function maxima()
    {
        return 1500;
    }
    public function media()
    {
        return 1200;
    }
    public function minima()
    {
        return 120;
    }
}
```

Fonte: Autoria própria, 2016.

Quadro 10. Exemplo de polimorfismo – Classe Carro

```
<?php
namespace Exemplos;

use Exemplos\Velocidade as Velocidade;

class Carro implements Velocidade
{
    public function maxima()
    {
        return 140;
    }
    public function media()
    {
        return 60;
    }
    public function minima()
    {
        return 15;
    }
}
```

Fonte: Autoria própria, 2016.

3.3 Padrões para codificação

Nas últimas versões do PHP foram implementados novos recursos como os namespaces, que cooperam na organização do código em uma hierarquia virtual, semelhantes aos pacotes de outras linguagens orientada a objetos. O SPL Autoload é um método com classes e interfaces padronizadas, com o objetivo de realizar o carregamento das classes automaticamente com os separadores de diretório do sistema hierárquico de arquivos (LOCKHART, 2015). Com o surgimento desses recursos, o PHP-FIG (The PHP Framework Interop Group) desenvolveu as PSRs (PHP Standard Recommendations), que são recomendações de padrões para codificação, envolvem o carregamento de classes, estrutura dos arquivos, nomenclatura de classes e métodos, sendo organizados em PSR-1, PSR-2, PSR-3 e PSR-4 (DALL'OGLIO, 2015).

A norma PSR-1 é responsável pelo estilo básico de codificação, é uma norma simples para ser implementada no código, para isso é necessário seguir alguns requisitos (PHP-FIG, 2016):

a) É obrigatório a utilização no código PHP as tags <?php ?> ou <?= ?>. Outras tags não são recomendadas.
b) A codificação dos arquivos PHP devem ser codificados com o conjunto de caracteres UTF-8 sem BOM (Marca de Ordem de Byte), essa opção está presente na maioria dos editores de texto e IDEs.

c) Um único arquivo PHP deve somente definir uma classe, ou executar uma determinada função. Não é recomendado um arquivo executar as duas tarefas.
d) Os namespaces e classes devem suportar o padrão autoload do PSR-4.
e) O nome de classes devem suportar o formato CamelCase, também conhecido como TitleCase. Exemplo: MinhaClasse.
f) Nome de constantes devem utilizar todos as letras em maiúsculas com underline. Exemplo: MINHA_CONSTANTE.
g) Os nomes de métodos devem usar o formato camelCase. Exemplo: meuMetodo.

A norma PSR-2 exemplificada no Quadro 11, é responsável por diretrizes mais rigorosas sobre codificação, com o objetivo de diminuir o mal entendimento do código por outros desenvolvedores, para isso é necessário aderir alguns requisitos importantes (PHP-FIG, 2016):

a) O código deve seguir a norma PSR-1.
b) O código deve ser indentado com quatro espaços em branco, sem a utilização da tecla Tab.
c) Os arquivos devem terminar com uma linha em branco, sem a tag ?>. Cada linha de código não pode exceder entre 80 e 120 caracteres.
d) Todas as palavras reservadas do PHP devem ser escritas em minúsculo. Exemplo: namespace, use, null, true, false.
e) Em cada declaração do namespace deve ser seguido por uma linha em branco.
f) Abrir chaves para classes é recomendado ir para a linha seguinte, para fechar deve-se ir na próxima linha após o fim da estrutura.
g) Abrir chaves para métodos é recomendado na mesma linha, para fechar logo após o fim da estrutura.
h) A visibilidade public, private e protected devem ser implementadas em todos os atributos e métodos, os abstratos e concretos devem ser declarados antes de sua visibilidade.
i) Estruturas como while, if, for, foreach, try ou catch devem ter um espaço logo a seguir, métodos e chamadas de função não devem ter espaços.
j) Abrir chaves para estruturas devem ser na mesma linha, e para fechar na próxima linha após o fim da estrutura.
k) Abrir e fechar parênteses em estruturas não devem ter espaços.

Quadro 11. Código seguindo a norma PSR-2

```php
<?php
namespace Vendor\Package;

use FooInterface;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class Foo extends Bar implements FooInterface
{
    public function sampleFunction($a, $b = null)
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // method body
    }
}
```

Fonte: PHP-FIG, 2016.

A norma PSR-3 ou LoggerInterface exemplificada no Quadro 12, é responsável pelos logs de um sistema PHP, um logger é um objeto que grava mensagens importantes para serem diagnosticadas, inspecionadas para solucionar problemas de estabilidade e desempenho do sistema (LOCKHART, 2015). A norma é prescrita utilizando uma interface que contém nove métodos que podem ser implementados, utilizando os níveis do protocolo syslog RFC 5424. (PHP-FIG, 2016).

Quadro 12. Exemplo da norma PSR-3 LoggerInterface

```php
<?php
namespace Psr\Log;

interface LoggerInterface
{
 public function emergency($message, array $context = array());
 public function alert($message, array $context = array());
 public function critical($message, array $context = array());
 public function error($message, array $context = array());
 public function warning($message, array $context = array());
 public function notice($message, array $context = array());
 public function info($message, array $context = array());
 public function debug($message, array $context = array());
 public function log($level, $message, array $context = array());
}
```

Fonte: LOCKHART, 2015.

A norma PSR-4 ou Autoloaders exemplificada no Quadro 13, descreve um autoload padronizado utilizando o método spl_autoload_register(), seu objetivo é carregar automaticamente as classes, interfaces ou arquivo de configuração de um sistema PHP, é levado para o interprete em tempo de execução para localizar o diretório do arquivo desejado. A norma PSR-0 era representada pelo método _autoload() do PHP, portanto esse método foi deprecado igualmente a norma (PHP-FIG, 2016). Antes do autoload ser implementado na linguagem, era necessário adicionar no topo dos arquivos os métodos include(), include_once(), require() e require_once() para carregar outros arquivos, portanto quando um sistema é de grande porte e existem milhares de arquivos para serem carregados, é nesta situação que surge a necessidade de utilizar o autoload padronizado (LOCKHART, 2015).

Quadro 13. Exemplo da norma PSR-4 Autoloaders

```
<?php

spl_autoload_register( function($cname)
{
    require_once ( getcwd() . DIRECTORY_SEPARATOR
    . str_replace('\\', '/', $cname) . '.php');
});
```

Fonte: Autoria própria, 2016.

3.4 Documentação para sistemas PHP

A documentação de sistemas é um hábito importante dentro do desenvolvimento de software, segundo Zandstra (2013) a documentação é importante principalmente em sistemas de grande porte, onde o código é extremamente extenso, quando uma documentação está presente facilita o entendimento das funcionalidades de cada método presente nos arquivos, em um exemplo, uma empresa que contém desenvolvedores experientes que conhecem cada atributo do sistema desenvolvido, ocorre a situação onde são contratados dois desenvolvedores para trabalhar neste sistema, podendo ser nas áreas de desenvolvimento ou manutenção, se este sistema não apresentar uma documentação, dificultará o entendimento das funções por parte dos novos desenvolvedores, portanto na condição onde se apresenta uma documentação bem formulada é possível diminuir o tempo gasto sobre a aprendizagem do sistema.

A criação de uma documentação com o PHP é uma tarefa fácil, podendo utilizar uma ferramenta que gera automaticamente todo o texto formatado para web e com uma boa aparência. O PHPDocumentor, é baseado em uma ferramenta desenvolvida em Java chamada JavaDoc. As

duas ferramentas extraem os comentários do código feitas pelo desenvolvedor de uma forma sofisticada, tendo no papel cada funcionalidade detalhada do sistema (ZANDSTRA, 2013).

Para utilizar o PHPDocumentor é necessário adquirir o arquivo phpDocumentor.phar disponível no endereço de seu site, encontrar a pasta de instalação do PHP, iniciar um prompt de comando dentro desta pasta, copiar o arquivo .phar e executar o comando com a sintaxe demonstrada no Quadro 14.

Quadro 14. Sintaxe para executar o PHPDocumentor

```
php.exe phpDocumentor.phar -d <arquivos-projeto> -t <pasta-de-saida>
 --title <titulo> --defaultpackagename <pacote> -p
```

Fonte: Autoria própria, 2016.

O Quadro 15 demonstra um trecho de código comentado com as suas respectivas tags conforme a ferramenta.

Quadro 15. Exemplo de documentação no código utilizando o PHPDocumentor

```php
<?php
/**
 * Recebe por injeção de dependência os componentes necessários.
 *
 * @param InterfaceConexao $db - Obrigatório uma conexão com o banco
 * @param View $view - Obrigatório uma view
 * @param string $Repositorio - Local das regras de negócio
 */
public function __construct(InterfaceConexao $db,View $view,
    $Repositorio)
{
    $this->db = $db;
    $this->view = $view;
    $this->Repositorio = $Repositorio;
}
```

Fonte: Autoria própria, 2016

Existem vários tipos de tags para serem utilizadas na documentação do PHP, cada uma tem sua importância dentro do PHPDocumentor, podendo ser possível registrar o autor principal do desenvolvimento do código, descrever a funcionalidade de cada classe, atributo ou método. Na Tabela 1 estão presentes algumas tags que foram utilizadas no desenvolvimento do sistema de gerenciador de biblioteca realizada no Capítulo 4.

Tabela 1. Tags do PHPDocumentor

Nome da tag	Descrição
@author	É utilizada para documentar o autor do sistema.
@package	Faz subdivisões lógicas, usando como suplemento para os Namespaces.
@param	Documenta um único argumento de um método.
@property	Permite que uma classe saiba quais propriedades mágicas estão presentes.
@return	Métodos que retornam algum valor.
@throws	Métodos que podem lançar uma exception.
@var	Documenta o tipo dos atributos e sua descrição.

Fonte: PHPDoc, 2016.

3.5 Frameworks

Conforme o autor Larman (2007), em modo geral um framework é um conjunto coeso de interfaces e classes que cooperam para fornecer funções para a parte básica de um sistema, em sua estrutura contém classes genéricas que implementam interfaces, incluindo interações entre objetos e oferecendo a liberdade para estender funções do framework, enfatizando a reutilização de software.

Para a linguagem PHP existem diversos tipos de frameworks, como as categorias Macro e Micro-Frameworks. As macros são conhecidas por oferecerem pacotes completos de componentes para o desenvolvimento, por exemplo: Zend Framework, Symfony ou Laravel. As micros são compostas por componentes específicos e de pequeno porte, umas são especialistas em rotas para sistemas, outras somente em arquitetura MVC, por exemplo: Slim ou Silex (LOCKHART, 2015). Atualmente os frameworks para PHP são padronizadas conforme a PSR, uma padronização da codificação e estrutura que foram estudados na sessão 3.3 deste Capítulo.

Com as vantagens que os frameworks oferecem para o desenvolvimento web, é importante destacar que mesmo com uma base ou arquitetura pronta, é possível desenvolver sistemas com falhas e sem padronização, contudo os desenvolvedores web precisam possuir o conhecimento

da engenharia de software e padrões de projeto para desenvolver sistemas em PHP com qualidade.

3.6 Principais problemas da linguagem PHP

O PHP é uma linguagem de script multiparadigma para web que abrange inúmeras formas de desenvolvimento, portanto essa linguagem é capaz de provocar alguns problemas para a elaboração de sistemas, muitas vezes sua simplicidade de codificação estrutural conduz os desenvolvedores diretamente para a fase de implementação do código, ignorando a etapa de concepção do projeto de software. Segundo o autor Bajgoric (2009), a facilidade de inserção de código PHP dentro de marcações HTML, que é uma linguagem de marcação utilizada para exibição de conteúdo em navegadores de internet, que acaba mesclando a apresentação com a lógica do sistema, gerando um código ilegível e de péssima manutenibilidade. Neste capítulo foram estudados os principais problemas que a linguagem PHP apresenta, como a junção de códigos HTML e PHP, e a programação seguindo o paradigma estrutural e orientado a objetos.

3.6.1 Combinação entre códigos HTML e PHP

O hábito de envolver em um único arquivo marcadores de cabeçalho ou rodapé com o código em PHP, pode gerar grandes problemas em aplicações de médio e grande porte, perdendo toda a lógica que envolve a separação de responsabilidades, misturando a lógica do negócio, consultas do banco de dados com a apresentação das informações. Em sistemas com essas características são propensos a duplicação de código, sendo difícil sua reutilização e manutenção (TRUCCHIA; ROMEI, 2010). Neste caso, o autor e desenvolvedor web Zervaas (2009, p. 03) aborda as razões para o PHP ser utilizado em larga escala.

> Uma das razões para o PHP se tornar tão popular é o fato de ser possível incluir código PHP diretamente dentro do código HTML que você está usando. Isso facilita o desenvolvimento de aplicações web simples e pequenas; em todo caso, isso tipicamente não pega bem. Quando um aplicação se torna grande, fica difícil também adicionar novas funcionalidades dentro de um grupo de marcadores HTML ou alterar o projeto do site por meio do deslocamento do código PHP (ZERVAAS, 2009, p. 03).

3.6.2 Programação estrutural X Orientada a objetos

A linguagem PHP primeiramente foi desenvolvida seguindo o paradigma de programação estrutural, linguagens antecessoras como Cobol, Pascal e C utilizavam esta estrutura que

permite a construção de um bloco constituído por instruções destinadas a executar uma tarefa, a estrutura de um sistema procedural possuí um conjunto numeroso de blocos que trabalham em conjunto (DALL’OGLIO, 2015). O PHP é uma linguagem acessível para qualquer nível de conhecimento em programação, portanto essa vantagem pode se transformar em desvantagem quando um sistema alcança níveis corporativos. É importante o conhecimento mais aprofundado sobre padrões de projeto e principalmente no paradigma de orientação a objetos, podendo ser possível atender os requisitos de sistemas de grande porte (TRUCCHIA; ROMEI, 2010).

A rotina estrutural em PHP pode promover problemas futuros para o desenvolvimento, o paradigma não contribui com uma organização modular da aplicação e nomes para os procedimentos, tornando-se responsabilidade do desenvolvedor adotar padrões individuais. Os procedimentos recebem somente parâmetros de entrada, realiza o processamento das informações e retornando um resultado, esse tipo de método não permite armazenar os dados para serem utilizados futuramente. Portanto o paradigma de programação estrutural não foi planejado para atender a modularidade de uma aplicação, o reuso de código e uma estrutura estável para suportar mudanças, com esses problemas os desenvolvedores do PHP construíram a linguagem seguindo o paradigma de orientação a objetos, no qual conserva-se em sua essência conceitos que cooperam para uma aplicação robusta (DALL’OGLIO, 2015).

4 MATERIAIS E MÉTODO

Um sistema de gerenciamento de biblioteca foi desenvolvido como um estudo de caso, utilizando o paradigma estrutural, foram elaboradas simulações comuns de mudanças realizadas em um sistema de acordo com novas necessidades, cada mudança necessitou da utilização de um padrão específico para serem resolvidas, podendo comprovar sua eficácia através do resultado de cada implementação e de uma eventual solução do problema, comparando as alterações realizadas e certificando-se que o problema inicial tenha sido corrigido, os resultados estão disponíveis no Capítulo 5. As versões completas dos sistemas desenvolvidos podem ser encontrados em um CD-ROM integrado com esta pesquisa.

A metodologia utilizada nesta pesquisa consiste em aplicar os padrões descritos no Capítulo 2 visando resolver os problemas da linguagem apresentados no Capítulo 3, através da refatoração de código do sistema estrutural e suas reescrita para adaptação do paradigma da orientação a objetos e o padrão arquitetural Model-View-Controller, sendo baseada em um modelo utilizado pelos autores Trucchia e Romei (2010).

4.1 Refatoração de código estrutural para orientado a objetos com MVC

A aplicação se trata de um gerenciador de biblioteca, onde há duas entidades, sendo Autor e Livro, cada uma desta entidade é representada por arquivos únicos no sistema estrutural, enquanto na versão MVC foram separadas em seus respectivos diretórios utilizando o recurso de namespaces e acompanhado por uma arquitetura padronizada. Com o objetivo de possibilitar a conversão do código estrutural para o MVC, foi desenvolvido uma pequena estrutura que pudesse atender aos requisitos da arquitetura demonstrado na Figura 7, para isso foi utilizado um controller genérico ou abstrato, que recebe por injeção de dependência componentes inicializados pela classe front controller, como as páginas da camada View, o repositório da camada Model que realiza o acesso ao banco de dados e as classes da camada Controller que são acessadas por meio de um sistema de rotas.

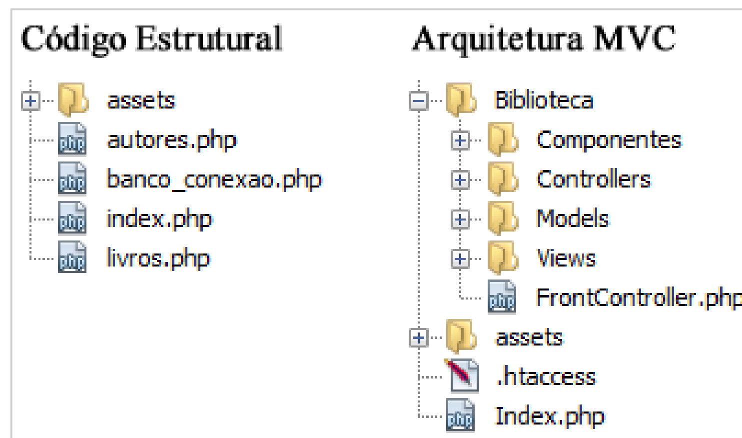

Figura 7. Estrutura hierárquica dos arquivos - Estrutural X MVC
Fonte: Autoria própria, 2016.

Para realizar a refatoração do sistema, foram utilizados os padrões de codificação PSR-1 e PSR-4, analisando o código inicial e sua estrutura, foi possível identificar a regra do negócio presente em vários arquivos diferentes e códigos duplicados, no arquivo autor.php do sistema estrutural apresentado no Quadro 16, foi possível encontrar as ações da entidade, estas ações foram identificadas como Controllers da arquitetura MVC, pois elas chamam as regras de negócio da aplicação e apresentam ao usuário a camada de apresentação.

Quadro 16. Trecho de código com as ações do sistema estrutural

```
<?php
switch(filter_input(INPUT_GET, 'acao'))
{
    case 'salvar':
        salvar();
        break;

    case 'excluir':
        excluir();
        break;

    case 'listar':
        listar();
        break;

    case 'ficha':
        ficha();
        break;
}
```

Fonte: Autoria própria, 2016.

Porém a entidade estrutural não faz essa separação, para isso foi necessário verificar-se quais métodos em comum todas as entidades possuem e criar-se uma interface genérica como demonstrado no Quadro 17, sendo baseado no padrão Estratégia e sugerido por Trucchia e Romei (2010).

Quadro 17. Trecho de código da interface de ações do sistema MVC

```
<?php
namespace Biblioteca\Controllers;

interface InterfaceAcoes
{
    public function listar();
    public function ficha($id);
    public function salvar();
    public function excluir($id);
}
```

Fonte: Autoria própria, 2016.

Com a interface desenvolvida, foi possível criar implementações de controllers usando esta interface para as entidades Autor e Livro, as ações podem ser acessadas através das URL’s do sistema, anteriormente no sistema estrutural como demonstrado no Quadro 18 onde realiza um caminho para a ficha cadastral de autores, eram utilizados diretamente os arquivos das entidades para ter acesso as ações.

Quadro 18. Trecho de código demonstrando a chamada de arquivos

```
<?php
// Função presente no arquivo AppEstrutural\autores.php

function listar()
{
    <div class="panel-footer">
        <a href="?p=autores&acao=ficha">Cadastrar novo autor</a>
    </div>
}
```

Fonte: Autoria própria, 2016.

Na versão refatorada foi possível criar um sistema de rotas para gerenciar as requisições através das URL’s demonstrado no Quadro 19, na classe front controller contém um método que constrói as rotas, onde se consiste em iniciar o sistema de rotas, que se responsabiliza por instanciar o controller correspondente com a classe, onde executa o método solicitado, informa um parâmetro e o despacha para o controller.

Quadro 19. Trecho de código com método de construir rotas

```
<?php
// Método presente no arquivo AppMvc\Biblioteca\FrontController.php

private function construirRotas()
{
    $router = new Router();

    $router->add('/^autores\/(\w+)\/?(\d+)?$/',function($metodo,$id=null)
    {
        $controller = $this->instanciarController
                               ('Biblioteca\Controllers\Autores');
        $controller->$metodo($id);
    });

    $router->despachar();
}
```

Fonte: Autoria própria, 2016.

O método demonstrado no Quadro 20, cria uma instancia do controller que recebe por injeção de dependência uma classe, onde retornará uma conexão com o banco de dados, uma página e o repositório. A classe controller que será instanciada estende a classe genérica do controller que contém esses mesmos parâmetros.

Quadro 20. Trecho de código com método que instancia um controller

```
<?php
// Método presente no arquivo AppMvc\Biblioteca\FrontController.php

private function instanciarController($classe)
{
    return new $classe($this->db, $this->view, $this->Repositorio);
}
```

Fonte: Autoria própria, 2016.

A classe controller de autores que foi instanciada pela classe front controller, ela implementa a interface de ações do Quadro 17, onde cria-se um objeto da classe Autor correspondente à camada Model da arquitetura MVC, relaciona o parâmetro com o código, busca as regras de negócio no repositório para listar os autores pelo código, estrutura o título da página e outras informações e exibe a página referente a ficha de autores por meio do sistema de views como mostra no Quadro 21.

Quadro 21. Trecho de código com método da ficha cadastral de autores

```
<?php
// Método presente no arquivo AppMvc\Biblioteca\Controllers\Autores.php

public function ficha($id)
{
    $autor = new Autor();
    $autor->cod_autor = $id;
    $autor = $this->Repositorio->listarAutorPorCod($autor);
    $this->view->vars['titulo'] = 'Ficha de autores';
    $this->view->vars['autor'] = $autor;
    $this->view->render('ViewSistema', 'AutoresFicha');
}
```

Fonte: Autoria própria, 2016.

As classes responsáveis por gerenciar as regras de negócio fazem parte da camada Model, foi criado uma interface para o repositório e a classe RepositorioPDO.php que implementa a interface para manipular as consultas do banco de dados, o Quadro 22 mostra um método para listar os autores pelo código, onde recebe por injeção de dependência a classe autor, retornando um método específico contendo a cláusula necessária para a consulta em SQL, sendo assim a classe, o nome da tabela, o código e a chave primária.

Quadro 22. Trecho de código com método que lista autores pelo código

```
<?php
// Método presente em AppMvc\Biblioteca\Models\RepositorioPDO.php
public function listarAutorPorCod(Autor $autor)
{
    return $this->selectById('Biblioteca\Models\Autor',
            'autores', 'cod_autor', $autor->cod_autor);
}
```

Fonte: Autoria própria, 2016.

Tendo as rotas, o controller instanciado com a conexão e repositório prontos, foi criado a camada View onde terá somente os códigos para as páginas em HTML, por meio do controller da entidade Autor demonstrado no Quadro 21, é possível exibir a página por meio das rotas utilizando somente o caminho para a ficha cadastral, sem utilizar diretamente o arquivo da entidade como apresentado no sistema estrutural no Quadro 23.

Quadro 23. Trecho de código em HTML com as rotas inclusas

```
<!--Arquivo presente em AppMvc\Views\Paginas\AutoresLista.php-->

<div class="panel-footer">
    <a href="/autores/ficha">Cadastrar novo autor</a>
</div>
```

Fonte: Autoria própria, 2016.

Com os dois sistemas desenvolvidos para esta pesquisa foi possível analisar que, o estrutural contém as regras de negócios acopladas com as páginas e funções, já o sistema com a arquitetura MVC foi possível desacoplar e gerenciar cada responsabilidade com as camadas criadas, facilitando o reuso e manutenção dos componentes e das classes desenvolvidas.

4.2 Implementando novos componentes

Com a aplicação da arquitetura MVC foi possível realizar modificações que suportam a troca de componentes, supondo que uma empresa utiliza o banco de dados MySQL e necessita de uma solução mais eficaz, com a análise da situação foi decidido implementar o banco de dados Microsoft SQL Server, no sistema de gerenciador de biblioteca estrutural não foi possível adicionar novos componentes, pois o sistema estrutural não permite a modularização, contudo seria necessário implementar novas ações no switch demonstrado no Quadro 16, aumentando sua complexidade e gastos de processamento.

Para demonstrar o processo da implementação de novos componentes, foi realizado uma simulação de uma possível situação de manutenção, onde é necessário adicionar outro driver de banco de dados, para isso foi criado uma nova classe de conexão para o SQL Server, na classe front controller, estão presentes as variáveis de configuração para ambos. O processo de configuração para troca do driver, é somente necessário mudar o valor da variável para “MSSQL” ou “MySQL” como mostra o Quadro 24.

Quadro 24. Configuração para troca de componentes

```
<?php
    // Driver a ser utilizado
    'driver' => 'MSSQL',

    // Configurações do driver do Microsoft SQL Server
    'MSSQL' => array
    (
        'servidor'  => 'localhost',
        'usuario'   => 'sa',
        'senha'     => 'root',
        'base'      => 'biblioteca'
    ),
    // Configurações do driver do MySQL Server
    'MySQL' => array
    (
        'servidor'  => 'localhost',
        'usuario'   => 'root',
        'senha'     => 'root',
        'base'      => 'biblioteca'
    )
```

Fonte: Autoria própria, 2016.

4.3 Modificando as regras de negócio

Com o sistema arquitetado com MVC foi possível fazer alterações na camada Model, supondo que as regras de negócio necessitam ser alteradas, é possível criar uma classe contendo a nova implementação sem substituir a anterior, podendo retorna-la caso seja necessário ou até mesmo alternar seu uso. No sistema estrutural não foi possível realizar as mudanças nas regras de negócio, para isso seria necessário implementar novas ações no switch apresentado no Quadro 16 e acoplar cada vez mais as funcionalidades dos arquivos no sistema.

Para demonstrar o processo de mudança, foi realizado uma simulação de uma possível situação de manutenção, onde as regras de negócio precisaram ser alteradas, onde o sistema gerenciador de biblioteca precisa exibir os nomes dos autores em maiúsculo conforme as normas da ABNT (Associação Brasileira de Normas Técnicas), para isso foi criado um novo repositório que estende o padrão, tendo seu método sobrescrito para atender a mudança e na classe front controller é possível modificar o repositório por meio das configurações de ambiente, alternando para “RepositorioPadrao” ou “RepositorioAbnt” como apresenta o Quadro 25.

Quadro 25. Configuração para troca de repositório

```
// Repositório a ser utilizado
'repositorio' => 'RepositorioAbnt';
```

Fonte: Autoria própria, 2016.

4.4 Gerando a documentação

De maneira como foi discutido no Capítulo 3, a documentação é um hábito fundamental para os processos de desenvolvimento e manutenção, portanto é necessário documentar os sistemas de gerenciador de biblioteca desenvolvidos para esta pesquisa, no caso o estrutural e o arquitetado com MVC. Para isso foi utilizado o PHPDocumentor, uma ferramenta útil para gerar automaticamente a documentação em formato de páginas web estilizadas.

O Quadro 26 mostra uma função do sistema estrutural para listar os autores, no início foi adicionado uma descrição da função e da variável utilizada seguindo as tags da ferramenta.

Quadro 26. Função comentada do sistema estrutural.

```php
<?php
/**
 * Função para listar o catálogo de autores.
 *
 * @global string $con Conexão.
 */
function listar()
{
    global $con;

    $q = "SELECT * FROM autores ORDER BY nome";
    $res = mysqli_query($con, $q);
    $autores = mysqli_fetch_all($res, MYSQLI_ASSOC);
}
```

Fonte: Autoria própria, 2016.

Para gerar a documentação é necessário executar o comando do PHPDocumentor apresentado no Capítulo 3, o Quadro 27 demonstra o comando com seus respectivos locais, título e pacote.

Quadro 27. Comando para gerar a documentação do sistema estrutural

```
php.exe phpDocumentor.phar -d "D:\Projetos\PHP\AppEstrutural" -t
"D:\Projetos\PHP\AppEstrutural\Docs" --title "Gerenciador de Biblioteca"
--defaultpackagename 'Biblioteca' -p
```

Fonte: Autoria própria, 2016.

Com a geração automática das informações para a documentação realizada pela ferramenta, logo em seguida foi salva no diretório do sistema estrutural. A Figura 8 demonstra o resultado final da documentação em uma página amigável.

Figura 8. Documentação do sistema estrutural.
Fonte: Autoria própria, 2016.

Para o sistema arquitetado com MVC foi utilizado a mesma ferramenta, o Quadro 28 apresenta um método protegido pertencente ao repositório do sistema, sendo responsável em selecionar os dados de uma tabela e ordenar pelo campo desejado, em seu início foi descrito sua funcionalidade, os parâmetros e o retorno, utilizando as tags do PHPDocumentor para especificar o tipo, o nome do atributo e sua descrição.

Quadro 28. Método comentado do sistema MVC.

```
<?php
/**
 * Método protegido responsável por selecionar todas os dados de uma
 * tabela qualquer do banco de dados.
 *
 * @param string $classe - Classe correspondente.
 * @param string $tabela - Tabela do Banco de dados.
 * @param string $order  - Cláusula ORDER BY.
 * @return array $stmt   - Retorna um statement com vários dados.
 */
protected function selectAll($classe, $tabela, $order)
{
    $stmt = $this->con->prepare("SELECT * FROM $tabela ORDER BY $order");
    $stmt->setFetchMode(PDO::FETCH_CLASS, $classe);
    $stmt->execute();
    return (array) $stmt->fetchAll();
}
```

Fonte: Autoria própria, 2016.

Para gerar a documentação do sistema MVC é necessário executar o comando do PHPDocumentor apresentado no Capítulo 3, o Quadro 29 demonstra o comando com seus respectivos locais, título e pacote.

Quadro 29. Comando para gerar a documentação do sistema MVC

```
php.exe phpDocumentor.phar -d "D:\Projetos\PHP\AppMvc" -t
"D:\Projetos\PHP\AppMvc\Docs" --title "Gerenciador de Biblioteca"
--defaultpackagename 'Biblioteca' -p
```

Fonte: Autoria própria, 2016.

Com a geração automática da documentação feita pelo comando, a Figura 9 demonstra o mesmo método apresentado no Quadro 28 com suas funcionalidades detalhadas. Também foi possível gerar um diagrama de classes do sistema por meio da ferramenta, estando presente no Apêndice A desta pesquisa.

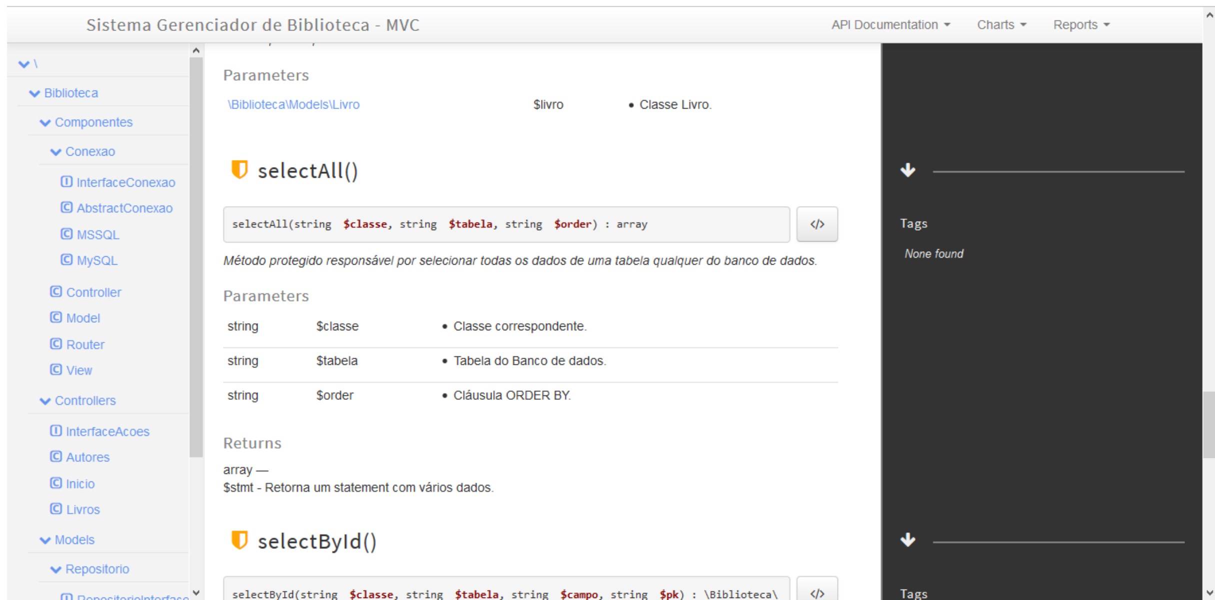

Figura 9. Documentação do sistema MVC
Fonte: Autoria própria, 2016.

A diferença entre as documentações são visíveis, a estrutural teve poucos dados apresentados sobre suas funcionalidades, todos os arquivos comentados foram mesclados em uma única página, dificultando a organização das informações, entretanto o sistema MVC apresentou uma excelente documentação, com dados ricos sobre cada funcionalidade de métodos, atributos e classes, os arquivos foram divididos por seus respectivos namespaces e packages do PHPDocumentor, formando uma hierarquia que possibilita a navegação entre os arquivos dentro da documentação. Porém é necessário implementar tags adicionais que serão importantes somente quando a documentação for gerada pela ferramenta, podendo deixar o código sobrecarregado de informações.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos no Capítulo 4, com o processo realizado de implementação de novos componentes, modificação das regras de negócio e geração da documentação do sistema estrutural e o arquitetado com MVC, que foram desenvolvidos para esta pesquisa.

A Tabela 2 demonstra as opções com suas descrições, que serão atribuídas aos resultados de cada sistema, onde o resultado efetivo significa que foi possível aplicar as simulações tendo efeitos positivos, os resultados parcialmente efetivos significa que foi possível a aplicação das simulações, porém com efeitos diferentes do esperado e o resultado não efetivo significa que foi possível a aplicação das simulações, porém apresentam efeitos negativos.

Tabela 2. Opções atribuídas aos resultados de cada sistema

Opções	Descrição
Sim	Resultado efetivo
Não	Resultado não efetivo
Parcial	Resultado parcialmente efetivo

Fonte: Autoria própria, 2016.

5.1 Resultado da implementação de novos componentes

Com a implementação de novos componentes nos sistemas desenvolvidos, foi possível extrair os seguintes resultados demonstrados na Tabela 3.

Tabela 3. Resultado da implementação de novos componentes

Modularização do sistema	Estrutural	MVC
Permite o uso de bancos de dados diferentes	Parcial	Sim
Permite a modularização de componentes	Não	Sim

Fonte: Autoria própria, 2016.

Os resultados mostram que o sistema arquitetado com MVC permite a modularização e o uso de vários bancos de dados, em comparação com o sistema estrutural, a estrutura permite parcialmente outros bancos de dados, ou seja, substitui o driver atual para outro, porém não é possível realizar a troca de drivers preservando o atual.

5.2 Resultado da modificação das regras de negócio

Foi realizada a modificação das regras de negócio nos sistemas desenvolvidos, foi possível extrair os seguintes resultados como mostra a Tabela 4.

Tabela 4. Resultado das mudanças de regras de negócio

Mudança de regras de negócio	Estrutural	MVC
Permitiu a mudança desacoplada da regra de negócio	Não	Sim
Gerou acoplamento nas novas alterações	Sim	Não

Fonte: Autoria própria, 2016.

Os resultados mostram que o sistema MVC permite mudanças nas regras de negócio sem gerar acoplamento entre seus componentes, no caso do sistema estrutural permite a mudança, mas para implementar as novas regras de negócio acabaria acoplando os arquivos e suas funcionalidades.

5.3 Resultado da geração da documentação

Foram adicionados comentários nos códigos e gerada a documentação de cada sistema utilizando a ferramenta PHPDocumentor, onde foi possível extrair os seguintes resultados como mostra a Tabela 5.

Tabela 5. Resultado da geração da documentação

Documentação	Estrutural	MVC
Permitiu a criação da documentação	Sim	Sim
Permitiu maiores detalhes da implementação	Não	Sim

Fonte: Autoria própria, 2016.

Os resultados mostram que a documentação foi gerada em ambos os sistemas, porém a documentação do sistema estrutural não apresenta detalhes suficientes para o entendimento completo de cada parte das funcionalidades presentes.

CONCLUSÃO

Com o estudo aprofundado da revisão bibliográfica realizada nos primeiros capítulos, foi possível adquirir o conhecimento necessário para apontar os principais problemas da linguagem PHP e aplicar os padrões essenciais nos sistemas desenvolvidos especialmente para esta pesquisa.

Utilizando-se a técnica de refatoração no sistema gerenciador de biblioteca, no Capítulo 4 foi realizado uma análise dos problemas encontrados no sistema estrutural e arquiteta-lo em um segundo sistema com o padrão Model-View-Controller sendo preservado as suas funcionalidades originais, com este sistema foi possível implementar novos componentes, modificar as regras de negócio e gerar uma documentação detalhada, onde os resultados foram apontados em tabelas apresentadas no Capítulo 5, com o objetivo de demonstrar a eficácia de cada implementação realizada no capítulo anterior.

Portanto com essas técnicas aplicadas, conclui-se que é possível otimizar o tempo de desenvolvimento de sistemas PHP e tempo gastos com a manutenção, onde é possível reutilizar uma grande parte dos componentes, podendo ser extensível para implementações futuras e tendo como maior vantagem, melhorar a qualidade do sistema, o ponto fundamental da engenharia de software.

REFERÊNCIAS BIBLIOGRÁFICAS

BAJGORIC, N. **Continuous computing technologies for enhancing business continuity**. 1 ed. Hershey: IGI Global, 2009.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. 3 ed. Westford: Addison-Wesley, 2013.

BELL, D. **The class diagram: an introduction to structure diagrams in UML 2**. **IBM**, 2004. Disponível em: <https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>. Acesso em: 28 set. 2016.

BOOCH, G. **Object-oriented analysis and design with applications**. 2 ed. Santa Clara: Addison-Wesley, 1994.

COLEMAN, A. **Playing with legos: the inner-working of a model-view-controller (MVC) web application**. **Self-Taught Coders**, 2016. Disponível em: <https://selftaughtcoders.com/model-view-controller-mvc-web-application/>. Acesso em: 26 set. 2016.

DALL'OGLIO, P. **PHP:** programando com orientação a objetos. 3 ed. São Paulo: Novatec, 2015.

FOWLER, M. **Padrões de arquitetura de aplicações corporativas**. 1 ed. Porto Alegre: Bookman, 2006.

GAMMA, E. et al. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. 1 ed. Porto Alegre: Bookman, 2007.

KERIEVSKY, J. **Refatoração para padrões**. 1 ed. Porto Alegre: Bookman, 2008.

LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3 ed. Porto Alegre: Bookman, 2007.

LOCKHART, J. **Modern PHP**: new features and good practices. 1 ed. Sabastopol: O'Reilly Media, 2015.

PRETTYMAN, S. **Learn PHP 7:** object-oriented modular programming using HTML5, CSS3, Javascript, XML, JSON and MySQL. 1 ed. Stone Mountain: Apress, 2016.

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

RODRIGUE, S. **Dependency injection in PHP**. **Envato Tuts**+, 2012. Disponível em: <https://code.tutsplus.com/tutorials/dependency-injection-in-php--net-28146>. Acesso em: 29 set. 2016.

SANDERS, W.B. **Learning PHP design patterns**. 1 ed. Sebastopol: O'Reilly Media, 2013.

SCHACH, S.R. **Object-oriented and classical software engineering**. 8 ed. New York: McGraw-Hill, 2011.

SOMMERVILLE, I. **Engenharia de software**. 9 ed. São Paulo: Pearson Prentice Hall, 2011.

PHPDoc. **Learn about phpDocumentor**. **PhpDocumentor**, 2016. Disponível em: <https://www.phpdoc.org/docs/latest/index.html>. Acesso em: 02 nov. 2016.

PHP-FIG. **PHP standards recommendations**. **The PHP Framework Interop Group**, 2016. Disponível em: <http://www.php-fig.org/psr/>. Acesso em: 04 out. 2016.

TRUCCHIA, F; ROMEI, J. **Pro PHP refactoring**. 1 ed. New York: Apress, 2010.

W3TECHS. **Usage of server-side programming languages for websites**. **W3Techs**, 2016. Disponível em: <http://w3techs.com/technologies/overview/programming_language/all>. Acesso em: 27 ago. 2016.

ZANDSTRA, M. **PHP objects, patterns, and practice**. 4 ed. New York: Apress, 2013.

ZERVAAS, Q. **Aplicações práticas de web 2.0 com PHP**. 1 ed. Rio de Janeiro: Alta Books, 2009.

APÊNDICE

APÊNDICE A

A Figura 10 apresenta um diagrama de classes do sistema gerenciador de biblioteca MVC gerado pela ferramenta PHPDocumentor, este diagrama está presente na documentação do sistema, incluso no CD-ROM acompanhado desta pesquisa.

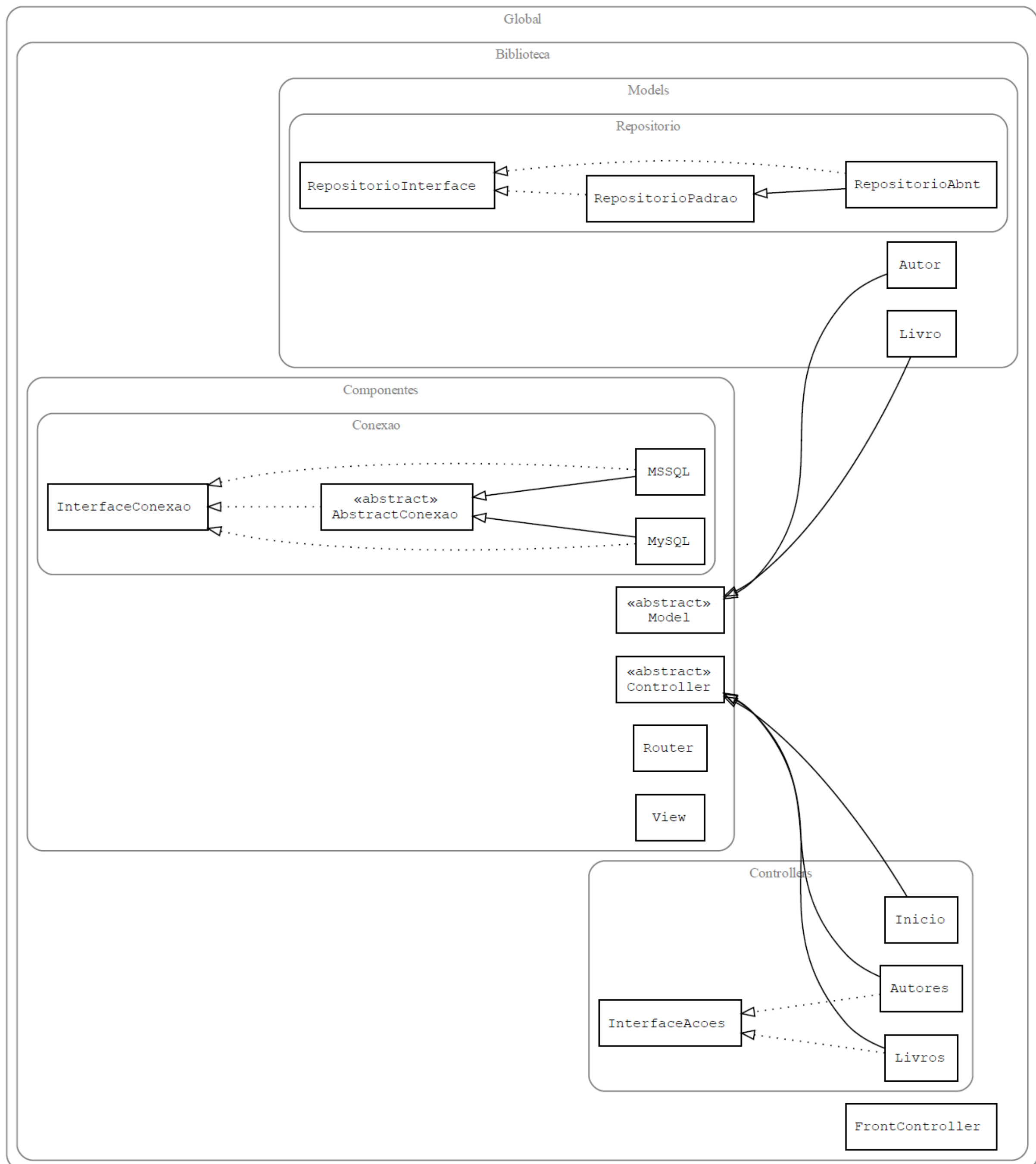

Figura 10. Diagrama de classes do sistema MVC
Fonte: Autoria própria, 2016.