

**CENTRO DE ENSINO SUPERIOR DE FOZ DO IGUAÇU  
CURSO DE CIÊNCIA DA COMPUTAÇÃO  
TRABALHO DE CURSO**

**PROPOSTA DE ARQUITETURA DE DESENVOLVIMENTO WEB  
BASEADA EM PHP UTILIZANDO DESIGN PATTERNS. UM ESTUDO  
DE CASO**

**FERNANDO GERALDO MANTOAN**

**FOZ DO IGUAÇU - PR**

**2009**

**FERNANDO GERALDO MANTOAN**

**PROPOSTA DE ARQUITETURA DE DESENVOLVIMENTO WEB  
BASEADA EM PHP UTILIZANDO DESIGN PATTERNS. UM ESTUDO  
DE CASO**

Trabalho de Conclusão de Curso submetido  
ao Centro de Ensino Superior de Foz do  
Iguaçu como parte dos requisitos para a  
obtenção do grau de bacharel em Ciência da  
Computação.

Orientador: Prof. Gildomiro Bairros

**FOZ DO IGUAÇU - PR**

**2009**

## **TERMO DE APROVAÇÃO**

**FERNANDO GERALDO MANTOAN**

### **PROPOSTA DE ARQUITETURA DE DESENVOLVIMENTO WEB BASEADA EM PHP UTILIZANDO DESIGN PATTERNS. UM ESTUDO DE CASO**

Este trabalho foi julgado adequado para a obtenção do grau de bacharel em Ciência da Computação e aprovado em sua forma final pelas disciplinas de Trabalho de Curso I e II.

#### **BANCA EXAMINADORA**

---

Prof. Arlete T. Beuren

---

Prof. Samuel Bellido

**FOZ DO IGUAÇU - PR**

**2009**

A todos os meus familiares, em especial à  
minha mãe por todo amor, apoio e carinho  
dado nesta longa jornada, esta conquista  
é a forma de demonstrar-lhe gratidão. À  
todos os meus amigos e à comunidade PHP.

# *Agradecimentos*

Primeiramente a Deus, por sempre me iluminar e ser o meu apoio nas horas mais difíceis, sempre sou muito grato pela ajuda que o Senhor me dá.

Aos meus pais, em especial à minha mãe, pelo incentivo e apoio, apesar de não parecer você sempre foi minha fonte de inspiração e motivação.

Também a todos os professores que estiveram nesta jornada de quatro anos, eles foram uma ótima fonte de conhecimento e tiveram suma importância na minha carreira acadêmica e profissional.

Agradeço também ao meu orientador, Prof. Gildomiro, que, além de ser um grande amigo e ex-colega de trabalho, também é um de meus professores, por todo apoio, idéias, paciência e amizade.

Agradeço aos meus colegas, pelos momentos de estudo, diversão e discussões.

*“Procure ser um homem de valor, em  
vez de ser um homem de sucesso.”*

***Albert Einstein***

# *Resumo*

A presente monografia apresenta a proposta de uma arquitetura de desenvolvimento web baseada em PHP que utiliza design patterns. A arquitetura foi construída utilizando Orientação a Objetos, o que permite a utilização dos diversos design patterns que são voltados à solução de problemas de projetos orientados a objetos. O principal objetivo da arquitetura é fornecer uma estrutura organizada, altamente reutilizável e produtiva; onde o ciclo de vida de uma aplicação pode ser prolongado, graças a alta manutenibilidade fornecida por esta arquitetura. Ao longo deste documento é feito um estudo sobre o conceito de arquiteturas de software, sobre design patterns e sobre o PHP e seus frameworks, além dele conter todos os detalhes da arquitetura definida e do estudo de caso de uma aplicação de controle de bibliotecas.

Palavras-chave: Arquitetura de Software, Design Patterns, PHP, Frameworks, Zend Framework.

# *Abstract*

The present monograph represents a proposal of a PHP based web development architecture that uses design patterns. The architecture was built using Object Oriented programming, which allows the use of the many existing design patterns that have the purpose of solving the object-oriented project problems. The main goal of the architecture is to provide an organized structure, highly reusable and productive; where the life cycle of an application can be increased, thanks to the high maintainability that the architecture provides. Through this document there will be a study about the concept of software architectures, design patterns and PHP and its frameworks, and furthermore, it will have every detail of the defined architecture and the case study of a library control application.

Key-words: Software Architecture, Design Patterns, PHP, Frameworks, Zend Framework.



# *Lista de Figuras*

2.1	Modelo em Cascata . . . . .	22
2.2	Modelo de 7 Camadas OSI . . . . .	23
3.1	Relacionamentos entre os padrões de projeto . . . . .	26
3.2	MVC Clássico . . . . .	36
3.3	<i>Data Mapper</i> . . . . .	37
3.4	<i>Table Data Gateway</i> . . . . .	38
4.1	Requisição CakePHP. . . . .	43
4.2	Requisição do <i>Code Igniter</i> . . . . .	45
6.1	Camadas da Arquitetura de <i>Software</i> . . . . .	53
7.1	Diagrama de Casos de Uso. . . . .	56
7.2	Diagrama de Classes. . . . .	58
7.3	Modelo de Entidades e Relacionamentos. . . . .	58
7.4	Estrutura de Diretórios. . . . .	59
7.5	Tela de listagem de empréstimos. . . . .	68
7.6	Tela de criação de um empréstimo. . . . .	69
7.7	Tela de devolução de item de empréstimo. . . . .	69

# *Lista de Tabelas*

7.1	Especificação dos casos de uso . . . . .	57
-----	--	----

# *Lista de Listagens*

7.1	Arquivo de Configuração . . . . .	59
7.2	<i>Facade</i> de Empréstimo . . . . .	60
7.3	<i>Factory</i> de <i>Facades</i> . . . . .	61
7.4	<i>Data Mapper</i> de Empréstimo . . . . .	62
7.5	<i>Table Data Gateway</i> de Empréstimo . . . . .	63
7.6	<i>Model</i> da entidade Empréstimo . . . . .	63
7.7	Classe <i>Observer</i> . . . . .	64
7.8	Classe <i>Observable</i> . . . . .	65
7.9	<i>Controller</i> de Empréstimo . . . . .	66
7.10	<i>View</i> de Listagem de Empréstimos . . . . .	67
7.11	Formulário de Empréstimo . . . . .	67

# *Lista de Abreviaturas e Siglas*

<b>AJAX</b>	<i>Asynchronous Javascript And XML</i>
<b>DRY</b>	<i>Don't Repeat Yourself</i>
<b>GOF</b>	<i>Gang of Four</i>
<b>HTML</b>	<i>Hypertext Markup Language</i>
<b>HTTP</b>	<i>Hypertext Transfer Protocol</i>
<b>JSON</b>	<i>JavaScript Object Notation</i>
<b>MVC</b>	<i>Model-View-Controller</i>
<b>PHP</b>	<i>Hypertext Preprocessor</i>
<b>PHP/FI</b>	<i>Personal Home Pages/Forms Interpreter</i>
<b>REST</b>	<i>Representational State Transfer</i>
<b>RSS</b>	<i>Really Simple Syndication</i>
<b>SGBD</b>	Sistema Gerenciador de Banco de Dados
<b>SOAP</b>	<i>Simple Object Access Protocol</i>
<b>SQL</b>	<i>Structured Query Language</i>
<b>UML</b>	<i>Unified Modeling Language</i>
<b>URL</b>	<i>Uniform Resource Locator</i>
<b>XML</b>	<i>Extensible Markup Language</i>
<b>XSS</b>	<i>Cross-site Scripting</i>

# Sumário

<b>1</b>	<b>Introdução</b>	<b>16</b>
1.1	Objetivos . . . . .	17
1.1.1	Geral . . . . .	17
1.1.2	Específicos . . . . .	17
1.2	Justificativa . . . . .	18
1.3	Metodologia . . . . .	18
<b>2</b>	<b>Arquiteturas de <i>Software</i></b>	<b>20</b>
2.1	A Importância de uma Arquitetura de <i>Software</i> . . . . .	21
2.2	Arquitetura de <i>Software</i> no processo de desenvolvimento de sistemas . . .	21
2.3	Arquitetura em Camadas . . . . .	23
<b>3</b>	<b><i>Design Patterns</i></b>	<b>25</b>
3.1	Padrões de Criação . . . . .	27
3.1.1	<i>Abstract Factory</i> . . . . .	27
3.1.2	<i>Builder</i> . . . . .	27
3.1.3	<i>Factory Method</i> . . . . .	27
3.1.4	<i>Prototype</i> . . . . .	28
3.1.5	<i>Singleton</i> . . . . .	28
3.2	Padrões Estruturais . . . . .	28
3.2.1	<i>Adapter</i> . . . . .	29
3.2.2	<i>Bridge</i> . . . . .	29

3.2.3	<i>Composite</i>	29
3.2.4	<i>Decorator</i>	30
3.2.5	<i>Facade</i>	30
3.2.6	<i>Flyweight</i>	30
3.2.7	<i>Proxy</i>	31
3.3	Padrões Comportamentais	31
3.3.1	<i>Chain of Responsibility</i>	31
3.3.2	<i>Command</i>	32
3.3.3	<i>Interpreter</i>	32
3.3.4	<i>Iterator</i>	32
3.3.5	<i>Mediator</i>	33
3.3.6	<i>Memento</i>	33
3.3.7	<i>Observer</i>	33
3.3.8	<i>State</i>	34
3.3.9	<i>Strategy</i>	34
3.3.10	<i>Template Method</i>	35
3.3.11	<i>Visitor</i>	35
3.4	<i>Design Patterns</i> Arquiteturais	36
3.4.1	<i>Model-View-Controller</i>	36
3.4.2	<i>Data Mapper</i>	37
3.4.3	<i>Table Data Gateway</i>	38
<b>4</b>	<b>PHP</b>	<b>39</b>
4.1	Histórico	39
4.2	<i>Frameworks</i>	40
4.2.1	<i>Zend Framework</i>	41
4.2.2	<i>CakePHP</i>	42

4.2.3	Prado . . . . .	44
4.2.4	<i>Code Igniter</i> . . . . .	44
<b>5</b>	<b>Ambiente Experimental</b>	<b>47</b>
5.1	Tecnologias Envolvidas . . . . .	47
5.1.1	UML . . . . .	47
5.1.2	PHP . . . . .	47
5.1.3	Apache HTTP . . . . .	48
5.1.4	HTML . . . . .	48
5.1.5	MySQL . . . . .	48
5.2	Padrões Envolvidos . . . . .	49
5.2.1	Programação Orientada a Objetos . . . . .	49
5.2.2	<i>Design Patterns</i> . . . . .	49
5.3	Estrutura Física . . . . .	50
5.3.1	Ambiente Físico . . . . .	50
5.3.2	Configurações de <i>Hardware</i> . . . . .	50
5.4	Estrutura Lógica . . . . .	51
5.4.1	Sistema Operacional . . . . .	51
5.4.2	Aplicativos . . . . .	51
5.4.2.1	Astah* Community . . . . .	51
5.4.2.2	Eclipse . . . . .	51
5.4.2.3	phpMyAdmin . . . . .	52
5.4.3	<i>Frameworks</i> . . . . .	52
5.4.3.1	Zend <i>Framework</i> . . . . .	52
<b>6</b>	<b>Arquitetura Definida</b>	<b>53</b>
6.1	Fluxo da Arquitetura . . . . .	53

<b>7</b>	<b>Implementação</b>	<b>55</b>
7.1	Documentação . . . . .	55
7.1.1	Casos de Uso . . . . .	55
7.1.2	Diagrama de Classes . . . . .	57
7.1.3	Banco de Dados . . . . .	58
7.2	Desenvolvimento da Aplicação . . . . .	59
7.2.1	Negócio . . . . .	60
7.2.1.1	<i>Facade</i> . . . . .	60
7.2.1.2	<i>Factory Method</i> . . . . .	61
7.2.1.3	<i>Data Mapper</i> e <i>Table Data Gateway</i> . . . . .	62
7.2.1.4	<i>Model</i> . . . . .	63
7.2.1.5	<i>Observer</i> . . . . .	64
7.2.2	Camada de Mais Alto Nível . . . . .	65
<b>8</b>	<b>Considerações Finais</b>	<b>70</b>
8.1	Trabalhos Futuros . . . . .	71
	<b>Referências Bibliográficas</b>	<b>72</b>



# 1 *Introdução*

Os padrões de projeto (*design patterns*), originaram-se na área de construção civil, onde Christopher Alexander e seus colegas proporam a idéia de utilizar uma linguagem padronizada para arquitetar edifícios e cidades. A *Gang of Four* (GOF), composta por 4 programadores: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides; viu que o conceito por trás dos padrões de projetos também se aplicava à área de informática, o que resultou em um livro com as especificações de vinte e três padrões de projeto para sistemas orientados a objeto. Os *design patterns* foram especificados para prover uma solução reutilizável a um problema de *software* comum a diversos projetos. (GAMMA et al., 1995)

O *Hypertext Preprocessor* (PHP) é uma das linguagens de *script* mais populares dos últimos anos. Um dos principais motivos desta popularidade é a baixa curva de aprendizagem. Esta facilidade da linguagem pode acarretar em códigos sem padronização. Com isso, quando um projeto PHP tem um aumento em sua complexidade e em seu tamanho, mais difícil de se dar manutenção ele se torna. Com a orientação a objetos, aprimorada na versão 5 da linguagem, foi possível se obter uma melhor codificação e reutilização de código. (HAYDER, 2007)

Além da orientação a objetos, pode-se aumentar ainda mais o ciclo de vida e a padronização de uma aplicação utilizando uma arquitetura de *software*. Segundo Bass, Clements e Kazman (2003), uma arquitetura de *software* é uma abstração de um sistema que suprime detalhes dos elementos que não afetam como eles utilizam, são utilizados por, se relacionam com, ou interagem com outros elementos.

Tendo estas premissas em vista, a proposta deste trabalho é fazer um estudo sobre os diversos *design patterns* especificados pela GOF, para propor uma arquitetura para desenvolvimento de aplicações *web* que contenha os *patterns* que auxiliem numa maior estrutura organizacional. Esta arquitetura será baseada em *frameworks* disponíveis para PHP, focando o princípio *Don't Repeat Yourself* (DRY) de se reaproveitar estruturas já

prontas para se trabalhar.

Procurando-se definir estes padrões de projeto e escolher os melhores *frameworks* para PHP pode-se levantar as seguintes questões: entre todos os *design patterns* existentes, como escolher os mais adequados, sem que haja um *anti-pattern* na arquitetura? Como modificar um *framework* PHP existente, aplicando-se *design patterns* para melhorar a organização do mesmo?

Para auxiliar na escolha dos *design patterns* é necessário uma análise nos *frameworks* que utilizam *design patterns* mais famosos da linguagem PHP e nos *design patterns* mais utilizados por desenvolvedores *web* experientes, separando-se os que mais se aplicarem à arquitetura proposta.

A escolha do *framework* que irá compor o núcleo da arquitetura será feita com base em um estudo entre os principais *frameworks Model-View-Controller* (MVC) feitos para a linguagem PHP, escolhendo-se o que mais oferece componentes, *helpers* e uma estrutura organizacional flexível.

## 1.1 Objetivos

### 1.1.1 Geral

Propor uma arquitetura de desenvolvimento de aplicações em PHP contendo *design patterns* que forneça uma maior estrutura organizacional, padronização de programação, facilidade de manutenção, menos repetição de código e que evite *bad smell*<sup>1</sup>.

### 1.1.2 Específicos

- Explicar os conceitos e a importância das arquiteturas de *software*;
- Explicar o conceito de *design patterns*, descrevendo os vinte e três *design patterns* definidos pela GOF;
- Apresentar a linguagem PHP e detalhar seus principais *frameworks*;
- Propor uma arquitetura de *software*, detalhando os *design patterns* e *frameworks* escolhidos para compô-la;

---

<sup>1</sup>Termo criado por Kent Beck e Martin Fowler, quando existe um “mal cheiro” no código isto significa que algo está errado, e que ele precisa ser refatorado, por exemplo, quando uma solução poderia ser simples e é implementada utilizando muitas linhas de código desnecessárias

- Realizar um estudo de caso na elaboração de uma aplicação de cadastro de livros, utilizando a arquitetura de *software* proposta.

## 1.2 Justificativa

“A importância de utilizar padrões se torna mais clara quando se passa a enxergar que um projeto pode se tornar inviável, seja por custo ou tempo, se o código tiver de ser escrito do zero toda vez e sem critérios.” (MELO; NASCIMENTO, 2007, p. 188)

Baseando-se em Gamma et al. (1995), os padrões de projeto tornam fácil a reutilização de projetos e arquiteturas, eles provêm alternativas de projeto que tornem um sistema reutilizável e que não comprometam a reusabilidade. Eles também aumentam a documentação e a manutenção de sistemas já prontos, por fornecerem uma especificação explícita de classes e interações de objetos e suas intenções subjacentes.

## 1.3 Metodologia

Quanto à natureza da pesquisa ela pode ser classificada como aplicada pois “[...] é feita a partir de objetivos que visam sua utilização prática. Valem-se essas pesquisas das contribuições das teorias e leis já existentes.” (PARRA; SANTOS, 2002, p.101)

Quanto aos objetivos, esta pesquisa classifica-se como exploratória pois “[...] têm como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses.” (GIL, 2002, p.41)

Tratando-se dos procedimentos de pesquisa será utilizado, inicialmente, a pesquisa bibliográfica, por ser “[...] desenvolvida com base em material já elaborado constituído principalmente de livros e artigos científicos.” (GIL, 2002, p.44). Posterior à etapa da pesquisa bibliográfica, será adotada a pesquisa experimental porque, segundo Gil (2002), consiste em determinar um objeto de estudo, selecionar as variáveis que seriam capazes de influenciá-lo, definir as formas de controle e de observação dos efeitos que a variável produz no objeto.

Na etapa de desenvolvimento do software de controle de estoque será utilizado o modelo de análise Orientado a Objetos, que segundo Hayder (2007, p.12) “[...] permite que um problema seja dividido em outros problemas menores que são comparativamente mais fáceis de compreender.”. Ao se fazer representação de dados, será utilizada a *Unified*

*Modeling Language* (UML). A UML é uma linguagem visual, criada durante a década de 1990, que visa fazer a modelagem de sistemas orientados a objeto, utilizando-se de vários elementos gráficos para construir diagramas que representem as várias visões de um sistema (MELO; NASCIMENTO, 2007).

## 2 *Arquiteturas de Software*

Uma arquitetura é o resultado de uma série de decisões de negócio e técnicas. Existem diversas influências para se trabalhar em um projeto, e a realização destas influências mudarão de acordo com o ambiente que a arquitetura deverá atuar. Um arquiteto projetando um sistema no qual os prazos sejam curtos fará uma série de escolhas, enquanto que o mesmo arquiteto projetando um sistema similar, porém com prazos maiores, poderá fazer diferentes escolhas para o projeto. Mesmo que os requisitos, *hardware*, *software* de suporte e recursos humanos disponíveis sejam parecidos, um projeto de *software* desenvolvido por um arquiteto nos dias atuais poderá ser diferente de um desenvolvido a cinco anos atrás. (BASS; CLEMENTS; KAZMAN, 2003)

A arquitetura é a representação que permite ao engenheiro de *software* analisar a efetividade do projeto em satisfazer seus requisitos declarados, considerar alternativas arquiteturais num estágio em que fazer modificações de projeto é ainda relativamente fácil e reduzir os riscos associados com a construção do software. (PRESSMAN, 2002)

Segundo Bass, Clements e Kazman (2003) uma arquitetura é, principalmente, uma abstração de um sistema que suprime detalhes de elementos que não afetam como eles utilizam, são utilizados por, se relacionam com ou interagem com outros elementos. Em quase todos os sistemas modernos, elementos interagem entre si a partir de *interfaces* que particionam detalhes sobre um elemento em partes públicas e privadas. A arquitetura está preocupada com a parte pública desta divisão; detalhes privados não são arquiteturais.

Tratando-se do projeto arquitetural, um componente de *software* pode ser tão simples quanto um módulo de um programa, mas também pode ser ampliado contendo bases de dados e *middleware* que permite a configuração de uma rede de clientes e servidores. As relações entre componentes podem ser simples como uma chamada de procedimento de um módulo, como também podem ser complexas como um protocolo de acesso a base de dados. (PRESSMAN, 2002)

## 2.1 A Importância de uma Arquitetura de *Software*

Para Bass, Clements e Kazman (2003) quando se trata da perspectiva técnica, existem três razões para a importância da arquitetura de *software*, que são:

- **Comunicação entre os envolvidos no projeto.** A arquitetura de *software* representa uma abstração comum de um sistema que muitos, se não todos, dos envolvidos no projeto podem utilizar como a base de um mútuo entendimento, negociação, consenso e comunicação;
- **Decisões de início de projeto.** Uma arquitetura de *software* manifesta as decisões iniciais de um sistema, estas possuindo profundo impacto com as próximas etapas, como o restante do desenvolvimento, o *deploy* do sistema, e o ciclo de manutenção. Este também é o ponto inicial onde decisões de projeto específicas sobre o sistema a ser desenvolvido podem ser analisadas;
- **Abstração transferível de um sistema.** A arquitetura constitui um modelo relativamente pequeno e intelectualmente inteligível de como um sistema é estruturado e como seus elementos trabalham em conjunto, e este modelo é transferível entre os sistemas. Em particular, ela pode ser aplicada a outros sistemas que exibam atributos de qualidade e requisitos funcionais similares, podendo promover reutilização em larga escala.

## 2.2 Arquitetura de *Software* no processo de desenvolvimento de sistemas

Segundo Sommerville (2003) no processo clássico de desenvolvimento de *software* chamado de modelo em cascata, existe uma sequência de fases a se seguir, onde cada fase depende de sua anterior. Ele pode ser visto na Figura 2.1.

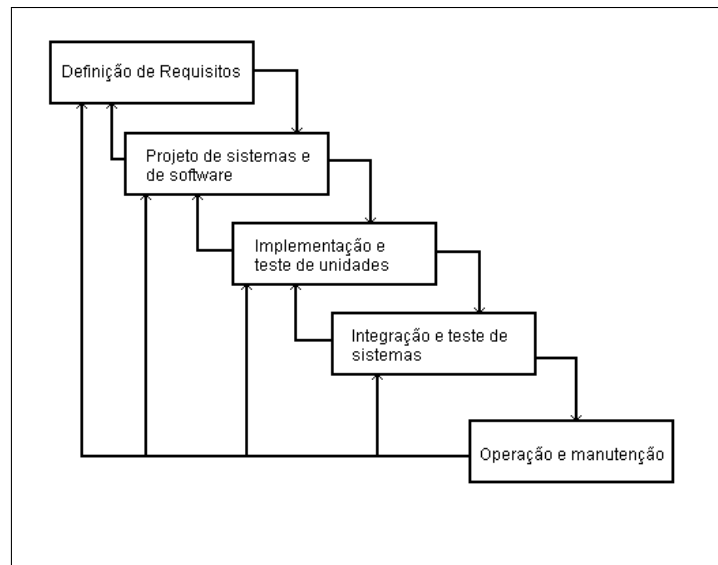


Figura 2.1: Modelo em Cascata  
(SOMMERVILLE, 2003)

Apesar da execução rigorosa das fases de engenharia de requisitos e de análise, é possível notar uma lacuna de informações a serem especificadas para que se possa prosseguir com a fase de projeto. Isto significa que especificar e modelar as funcionalidades de um sistema não é o suficiente para saber como o sistema deve ser estruturado e organizado, visando atender aos requisitos funcionais e atributos de qualidade. (VAROTO, 2002)

A arquitetura de *software* propõe diversas atividades que tentam cobrir esta lacuna entre as fases de análise e projeto, dentre elas a elaboração de um modelo de domínio que ressalta o escopo do sistema, a identificação das dependências de construção e o mapeamento dos requisitos não funcionais que o sistema deve atender e que não foram totalmente especificados na fase de engenharia de requisitos. A diferença entre as fases de análise e de projeto e as atividades de arquitetura são evidentes quando se trata de sistemas grandes e complexos, principalmente por, nestes casos, o entendimento claro, o escopo e as possíveis evoluções serem mais difíceis de identificar, dado o tamanho da incerteza resultante da falta de conhecimento do negócio. (VAROTO, 2002)

A construção da arquitetura deve objetivar o sistema como um todo, mas com os elementos mínimos necessários para realizar a implementação da primeira versão. Se a arquitetura for focada apenas nas funcionalidades priorizadas para a primeira versão, a incorporação de mudanças ou novas funcionalidades para a próxima versão pode exigir uma alteração tão grande que seja necessário refazer toda a arquitetura, implicando em tempo e custos adicionais. (VAROTO, 2002)

## 2.3 Arquitetura em Camadas

Em uma arquitetura em camadas um certo número de camadas diferentes são definidas, cada uma realizando operações que se tornam progressivamente mais próximas do conjunto de instruções de máquina. Na camada exterior, os componentes operam na *interface* com o usuário. Na camada mais interna os componentes realizam a *interface* com o sistema operacional. As camadas intermediárias fornecem serviços utilitários e funções do *software* de aplicação. (PRESSMAN, 2002)

Protocolos de rede são os exemplos mais conhecidos de arquitetura em camadas. Cada protocolo consiste em uma série de regras e convenções que descrevem como programas de computadores se comunicam sobre as fronteiras de máquinas. O formato, conteúdo e significado de todas as mensagens são definidos. Todos os cenários são descritos em detalhes, geralmente através de gráficos de sequência. O protocolo especifica acordos em uma variedade de camadas de abstração, indo desde detalhes da transmissão de bits até a lógica da aplicação de mais alto nível. Portanto projetistas usam diversos subprotocolos e organizam eles em camadas. Cada camada lida com um aspecto específico de comunicação e usa o serviço da próxima camada mais baixa. Um exemplo de utilização de arquitetura em camadas é o Modelo de 7 Camadas OSI, utilizado em redes de computadores, este é apresentado na Figura 2.2. (BUSCHMANN et al., 1996)

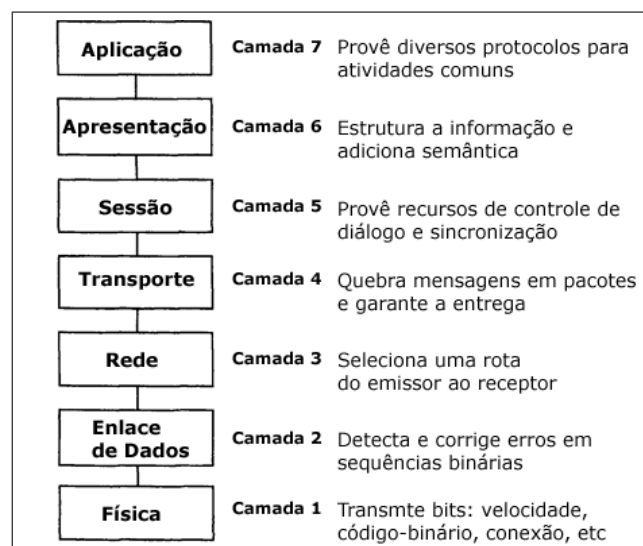


Figura 2.2: Modelo de 7 Camadas OSI

(BUSCHMANN et al., 1996 apud TANENBAUM, 1992) Adaptada.

Uma abordagem em camadas é considerada de melhor prática do que a implementação



do protocolo como um bloco monolítico<sup>1</sup>, porque a implementação separada de problemas que possuem conceitos distintos resulta em diversos benefícios, como, por exemplo, possibilidade de desenvolvimento em equipes. A utilização de partes semi-independentes também fornece a possibilidade de se trocar com mais facilidade partes individuais posteriormente. Melhores tecnologias de implementação como novas linguagens ou algoritmos podem ser incorporadas, simplesmente re-escrevendo uma seção de código delimitada. (BUSCHMANN et al., 1996)

---

<sup>1</sup>diz-se dos elementos que formam um todo rígido, homogêneo.

### 3 *Design Patterns*

Segundo Gamma et al. (apud ALEXANDER et al., 1977), cada padrão de projeto (*design pattern*) descreve um problema que ocorre constantemente no ambiente de trabalho, e então descreve o núcleo da solução para aquele problema, de uma maneira que possa-se utilizar esta solução milhares de vezes, nunca precisando fazer a mesma coisa mais de uma vez.

Este conceito definido por Alexander era direcionado a padrões de construção civil, porém ele também se torna verdadeiro em projetos de *software* orientados a objeto. As soluções são expressas em termos de objetos e *interfaces* ao invés de paredes e portas, mas no núcleo dos dois tipos de padrões encontra-se uma solução de um problema em um determinado contexto. (GAMMA et al., 1995).

Assim, segundo Gamma et al. (1995), um padrão de projeto nomeia, abstrai e identifica os principais aspectos de uma estrutura de projeto comum, tornando esta estrutura útil para a criação de um projeto que seja orientado a objetos e reutilizável. O padrão de projeto identifica as classes e instâncias participantes, suas regras e colaborações e a distribuição de responsabilidades. Cada padrão de projeto está focado em um problema de projeto orientado a objeto particular. Ele descreve quando é aplicado, se pode ser aplicado visando outras restrições do projeto e as consequências de seu uso.

Padrões de projeto tornam fácil de se reutilizar projetos e arquiteturas e ajudam na escolha de alternativas de projetos que tornam um sistema reutilizável e evitam alternativas que comprometem a reusabilidade. Eles podem ainda melhorar a documentação e manutenção de sistemas existentes por eles proporcionarem uma especificação explícita de classes e interações de objetos e suas intenções subjacentes. Apresentando de uma maneira mais simples, padrões de projeto ajudam um projetista a definir um projeto muito mais rapidamente. (GAMMA et al., 1995)

Os padrões de projeto possuem formas de classificação a partir de seu propósito que, segundo Gamma et al. (1995), pode ser de **criação**, de **estruturação** ou de **comporta-**



## 3.1 Padrões de Criação

Os padrões de projeto de criação abstraem o processo de instanciação. Eles ajudam a fazer com que um sistema seja independente de como seus objetos são criados, compostos e representados. Um padrão de criação de classes utiliza herança para variar a classe que é instanciada, enquanto que um padrão de criação de objetos delegará a instanciação a outro objeto. (GAMMA et al., 1995)

### 3.1.1 *Abstract Factory*

O intuito do padrão *Abstract Factory* é prover uma *interface* para a criação de famílias de objetos relacionados ou dependentes entre si, sem a necessidade de se especificar suas classes concretas. (GAMMA et al., 1995)

Um dos principais benefícios que o *Abstract Factory* traz é seu auxílio no controle de classes e objetos que uma aplicação cria. Devido a uma *factory* encapsular a responsabilidade e o processo de criação de objetos, ela isola clientes de implementar classes. Eles manipularão instâncias através de suas *interfaces* abstratas. (GAMMA et al., 1995)

### 3.1.2 *Builder*

Segundo Gamma et al. (1995), o objetivo do padrão de projeto *Builder* é separar a construção de um objeto complexo de sua representação, fazendo com que o mesmo processo de construção possa criar diferentes representações.

O *Builder* melhora a modularidade por encapsular a forma com que um objeto complexo é construído e representado. Os clientes não precisam saber nada sobre as classes que definem a estrutura interna de um produto, estas classes não aparecem na *interface* deste padrão de projeto. Cada *Builder* concreto contém o código para criar e montar um tipo particular de produto. (GAMMA et al., 1995)

### 3.1.3 *Factory Method*

Este padrão de projeto define uma *interface* para a criação de um objeto, mas deixa que sub-classes decidam qual classe instanciar. O *Factory Method* deixa que uma classe delegue a instanciação para as subclasses. (GAMMA et al., 1995)

Segundo Gamma et al. (1995), ao utilizar o *Factory Method* pode-se eliminar a neces-

sidade de se ligar classes específicas da aplicação ao código. O código apenas lidará com a *interface* da classe; portanto ele poderá trabalhar com qualquer classe concreta definida pelo usuário.

### 3.1.4 *Prototype*

O *Prototype* especifica os tipos de objetos a se criar utilizando uma instância como protótipo, e cria novos objetos copiando-se este protótipo. (GAMMA et al., 1995)

Assim como os padrões *Abstract Factory* e *Builder*, o *Prototype* encapsula as classes concretas do cliente, dessa forma reduzindo o número de nomes que os clientes conhecem. Além disso, estes padrões permitem que um cliente trabalhe com classes específicas da aplicação sem modificações. (GAMMA et al., 1995)

### 3.1.5 *Singleton*

O principal objetivo do padrão de projeto *Singleton*, segundo Gamma et al. (1995), é garantir que uma classe tenha somente uma instância, e forneça um ponto global de acesso a ela.

Alguns dos benefícios que este padrão traz são:

- **Acesso controlado a uma única instância:** Devido à classe *Singleton* encapsular sua única instância, pode-se obter um controle restrito em torno de como e quando clientes a acessarão;
- **Espaço de nomes reduzido:** O padrão *Singleton* é uma melhoria de variáveis globais. Ele evita a poluição do espaço de nomes com variáveis globais que armazenam instâncias únicas. (GAMMA et al., 1995)

## 3.2 Padrões Estruturais

Segundo Gamma et al. (1995), padrões classificados como estruturais estão preocupados em como classes e objetos são compostos para formar estruturas maiores. Padrões estruturais de classe usam herança para compor *interfaces* ou implementações. Padrões estruturais de objeto descrevem caminhos para compor objetos para se chegar a novas funcionalidades. A flexibilidade de composição de objetos é possível devido à habilidade

de se mudar a composição em tempo de execução, o que é impossível em uma composição de classes estáticas.

### 3.2.1 *Adapter*

*Adapter*, é um padrão de projeto que tem como principal objetivo converter a *interface* de uma classe em uma outra *interface* esperada por clientes. O *Adapter* permite que classes que não poderiam trabalhar em conjunto, devido à incompatibilidade de suas *interfaces*, se comuniquem entre si. (GAMMA et al., 1995)

Uma classe adaptadora possibilita a sobrescrita de métodos da classe a ser adaptada, devido a uma adaptadora extendê-la. Ela também inicia somente um objeto, e nenhum apontador adicional é necessário para se obter a classe adaptada. (GAMMA et al., 1995)

Um objeto adaptador permite a um único *Adapter* a possibilidade de se trabalhar com várias classes adaptadas, ou seja, a própria classe que deve ser adaptada e todas as suas subclasses (se elas existirem). O *Adapter* também pode adicionar funcionalidades para todas as classes adaptadas de uma vez. (GAMMA et al., 1995)

### 3.2.2 *Bridge*

O intuito do *Bridge*, segundo Gamma et al. (1995), é desacoplar uma abstração de sua implementação para que as duas possam variar independentemente.

Entre seus principais benefícios, o *Bridge* traz uma melhoria significativa de extensibilidade. Pode-se estender a hierarquia de Abstração e Implementação independentemente. Ele também garante que uma implementação não esteja ligada permanentemente a uma *interface*. A implementação de uma abstração pode ser configurada em tempo de execução. (GAMMA et al., 1995)

### 3.2.3 *Composite*

O padrão *Composite* tem como objetivo, baseando-se em Gamma et al. (1995), compor objetos em estruturas de árvore para representar hierarquias parcialmente completas. O *Composite* permite que clientes tratem objetos individuais e composições de objetos uniformemente.

Uma das consequências de se utilizar o padrão *Composite*, segundo Gamma et al. (1995), é a hierarquia de classes que ele fornece, sendo esta hierarquia composta de objetos

primitivos e objetos compostos. Os objetos primitivos podem ser utilizados para compor objetos mais complexos, que também podem formar outras composições, e assim por diante. Onde quer que o código cliente espere um objeto primitivo, ele também pode pegar um objeto composto.

### 3.2.4 *Decorator*

Segundo Gamma et al. (1995), o principal objetivo do padrão *Decorator* é incorporar responsabilidades adicionais a um objeto dinamicamente. Objetos “decoradores” fornecem uma alternativa flexível à utilização de heranças, para se estender funcionalidades.

O padrão *Decorator* fornece uma maneira mais flexível para se adicionar responsabilidades a objetos do que a que se pode ter com herança estática (múltipla). Com decoradores, responsabilidades podem ser adicionadas e removidas em tempo de execução simplesmente incorporando e desincorporando-as. Em contraste, para se realizar a herança é necessária a criação de uma nova classe para cada responsabilidade adicional. Isso traz o crescimento de classes e aumenta a complexidade de um sistema. (GAMMA et al., 1995)

### 3.2.5 *Facade*

O padrão de projeto *Facade* fornece uma *interface* única para um conjunto de *interfaces* de um subsistema. Uma *Facade* define uma *interface* de mais alto nível que torne o subsistema mais fácil de ser utilizado. (GAMMA et al., 1995)

Um dos benefícios que o padrão *Facade* fornece é a proteção dos componentes de um subsistema, abstraindo-o dos códigos clientes, dessa forma ele reduz o número de objetos que clientes precisam lidar e torna o subsistema mais fácil de se utilizar. Outro benefício é que ele diminui o acoplamento entre o subsistema e seus clientes. Constantemente os componentes em um subsistema são fortemente acoplados. (GAMMA et al., 1995)

### 3.2.6 *Flyweight*

Este padrão utiliza-se de compartilhamento para suportar de maneira eficiente um grande número de objetos de granularidade fina. (GAMMA et al., 1995)

*Flyweights* podem trazer custos em tempo de execução associados a transferência, busca, e outros estados extrínsecos de computação, especialmente se ele antigamente

foi guardado como um estado intrínseco. No entanto, estes custos são compensados com economias no espaço, que aumentam quanto mais *Flyweights* são compartilhados. (GAMMA et al., 1995)

### 3.2.7 *Proxy*

O intuito do padrão *Proxy*, segundo Gamma et al. (1995), é prover um substituto ou marcador de localização de outro objeto para controlá-lo e acessá-lo.

O padrão *Proxy* provê uma camada indireta para quando for feito o acesso a um objeto. Ele também possui outra otimização para esconder do cliente, ela é chamada de “**copie-na-escrita**” e é relacionada a criação de acordo com a demanda. Copiar um objeto grande e complicado pode ser uma operação árdua. Se a cópia nunca é modificada, então não existe necessidade deste custo ocorrer. Ao se utilizar um *proxy* para retardar o processo de cópia, garante-se que se pague o preço da cópia deste objeto somente quando ele for modificado. (GAMMA et al., 1995)

## 3.3 Padrões Comportamentais

Padrões comportamentais estão preocupados com algoritmos e com a atribuição de responsabilidades entre objetos. Padrões comportamentais descrevem não somente padrões de objetos ou classes mas também os padrões de comunicação entre eles. Estes padrões caracterizam controle complexo de fluxo que é difícil de se acompanhar em tempo de execução. Eles tiram o foco do controle de fluxo para que possa-se concentrar somente na maneira como objetos são interconectados. (GAMMA et al., 1995)

### 3.3.1 *Chain of Responsibility*

O padrão *Chain of Responsibility* tem como objetivo evitar o acoplamento do remetente de uma requisição com seu destinatário ao dar chances de se tratar a requisição a mais de um objeto. Ele cria uma corrente com os objetos recebedores e passa a requisição para a corrente até que um objeto manipule-a. (GAMMA et al., 1995)

Com o *Chain of Responsibility* pode-se reduzir o acoplamento. Este padrão libera um objeto de conhecer quais outros objetos manipulam uma requisição. Um objeto apenas deve saber que uma requisição será manipulada adequadamente. O padrão também adiciona mais flexibilidade por atribuir responsabilidades entre os objetos. Pode-se adicionar



ou modificar responsabilidades para a manipulação de uma requisição por adicionar ou modificar a corrente em tempo de execução. (GAMMA et al., 1995)

### 3.3.2 *Command*

O principal intuito do *Command* é encapsular uma requisição como um objeto, dessa forma deixando que os clientes parametrizem diferentes requisições, filas de espera ou requisições de *log*, também dando suporte a operações que podem ser desfeitas. (GAMMA et al., 1995)

O padrão *Command* possui as seguintes consequências:

- Ele desacopla o objeto que invoca a operação do objeto que sabe como executá-la;
- *Commands* são objetos de primeira classe. Eles podem ser manipulados e estendidos assim como qualquer outro objeto;
- Adicionar novos *Commands* é uma tarefa fácil, porque não é necessária a modificação de classes já existentes. (GAMMA et al., 1995)

### 3.3.3 *Interpreter*

Dada uma linguagem, o padrão *Interpreter* define uma representação para a sua gramática juntamente com um interpretador que utiliza a representação para interpretar sentenças na linguagem. (GAMMA et al., 1995)

Entre os principais benefícios do padrão *Interpreter*, segundo Gamma et al. (1995), pode-se destacar a facilidade de se modificar e estender a gramática de uma linguagem, devido ao padrão utilizar classes para representar regras de gramática, pode-se utilizar herança para modificar ou estender a gramática. Expressões existentes podem ser modificadas incrementalmente, e novas expressões podem ser definidas como variações de expressões antigas.

### 3.3.4 *Iterator*

Segundo Gamma et al. (1995), este padrão provê uma forma de acessar elementos de um objeto agregado sequencialmente sem expor a representação interna deste objeto.

O padrão *Iterator* simplifica uma *interface* agregada. Sua *interface* travessa evidencia a necessidade de uma *interface* similar na agregação, desta forma simplificando a *interface* agregada. (GAMMA et al., 1995)

### 3.3.5 *Mediator*

O objetivo do padrão *Mediator*, segundo Gamma et al. (1995), é definir um objeto que encapsula a forma como um conjunto de objetos interage. O *Mediator* promove um fraco acoplamento por evitar que objetos refiram-se uns aos outros explicitamente, e ele permite que a interação destes objetos possa variar de maneira independente.

Este padrão limita a criação de subclasses, pois ele localiza comportamentos que de outra forma seriam distribuídos através de diversos objetos. A modificação deste comportamento requer que as subclasses estendam o *Mediator*. O *Mediator* também simplifica protocolos de objeto, ele sobrescreve interações muitos-para-muitos com interações um-para-muitos entre o mediador e seus “colegas”. (GAMMA et al., 1995)

### 3.3.6 *Memento*

Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto para que, futuramente, o objeto possa ser restaurado para este estado. (GAMMA et al., 1995)

O *Memento* evita a exposição de informações que somente um “originador” deve gerenciar mas que no entanto devem ser armazenadas fora do originador. O padrão protege outros objetos da complexidade interna do *Originator*, preservando desta forma os limites da encapsulação. (GAMMA et al., 1995)

### 3.3.7 *Observer*

O padrão *Observer* define uma dependência um-para-muitos entre objetos para que quando o estado de um objeto seja modificado, todos os seus dependentes sejam notificados e atualizados automaticamente. (GAMMA et al., 1995)

Com este padrão é possível modificar de forma independente assuntos e observadores. Pode-se reutilizar assuntos sem reutilizar seus observadores e vice-versa. Ele permite que se adicione observadores sem modificar o assunto ou outros observadores. (GAMMA et al., 1995)

Outro benefício de sua utilização é o acoplamento abstrato entre *Subject* e *Observer*, ou seja, tudo que um assunto sabe é que ele possui uma lista de observadores, cada um estando nos conformes da *interface* simples da classe abstrata *Observer*. O assunto não sabe a classe concreta de nenhum dos observadores. Assim o acoplamento entre assuntos e observadores é abstrato e mínimo. (GAMMA et al., 1995)

### 3.3.8 *State*

O padrão de projeto *State*, segundo Gamma et al. (1995), permite que um objeto altere seu comportamento quando seu estado interno for modificado. O objeto aparecerá para modificar sua classe.

O padrão *State* coloca em um objeto todos os comportamentos associados com um estado particular. Devido a todo o código específico de estado estar em uma subclasse de um *State*, novos estados e transições podem ser adicionados facilmente através da definição de novas subclasses. (GAMMA et al., 1995)

Ele também faz com que transições de estado sejam explícitas. Quando um objeto define seu estado atual a partir de valores de dados internos, suas transições de estados não possuem representações explícitas; elas apenas aparecem como definições a algumas variáveis. As transições se tornam mais explícitas ao se introduzir objetos separados para diferentes estados. (GAMMA et al., 1995)

### 3.3.9 *Strategy*

Baseando-se em Gamma et al. (1995), o *Strategy* define uma família de algoritmos, encapsula cada um, e faz com que eles sejam intercambiáveis. Ele permite que o algoritmo seja alterado independentemente pelos clientes que o utilizam.

O *Strategy* define famílias de algoritmos relacionados. Hierarquias de classes *Strategy* definem uma família de algoritmos ou comportamentos de diferentes contextos para reutilizá-la. A Herança pode ajudar a se retirar funcionalidades comuns dos algoritmos. (GAMMA et al., 1995)

Segundo Gamma et al. (1995), ele também fornece uma maneira alternativa à subclasses. A herança fornece outra maneira de suportar uma variedade de algoritmos ou comportamentos. Porém ela pode misturar a implementação do algoritmo de uma classe, tornando-a mais difícil de se entender, manter, e estender. Também não se pode modificar

o algoritmo dinamicamente. Ao se encapsular o algoritmo em classes *Strategy* separadas, pode-se alterar o algoritmo independentemente de seu contexto, tornando-o fácil de se modificar, entender e estender.

### 3.3.10 *Template Method*

O *Template Method* define o esqueleto de um algoritmo em uma operação, permitindo que subclasses possam, posteriormente, prover funcionalidades específicas. Ele permite que subclasses redefinam certas etapas de um algoritmo sem modificar a estrutura dele. (GAMMA et al., 1995)

*Template Method* é uma técnica fundamental para a reutilização de código. Eles são particularmente importantes em bibliotecas de classes, porque eles são o meio de se fatorar comportamentos comuns em uma biblioteca de classes. Este padrão faz com que se tenha um controle de estrutura inverso, ou seja, a maneira como uma classe pai chama a operação de uma subclasse e não o contrário. (GAMMA et al., 1995)

É importante para *Template Methods* especificar quais operações **podem** ser sobrescritas e quais operações **devem** ser sobrescritas. Para se reutilizar efetivamente uma classe abstrata, escritores de subclasses devem entender quais operações são projetadas para a reescrita. (GAMMA et al., 1995)

### 3.3.11 *Visitor*

O padrão de projeto *Visitor*, baseando-se em Gamma et al. (1995), representa uma operação a ser executada sobre os elementos de uma estrutura de objetos. O *Visitor* permite a definição de uma nova operação sem modificar as classes dos elementos sobre os quais opera.

Com este padrão, facilita-se a adição de operações que dependam dos componentes de objetos complexos. Pode-se definir uma nova operação sobre a estrutura de um objeto simplesmente adicionando um novo *Visitor*. Em contraste, se a funcionalidade é espalhada em diversas classes, então deve-se modificar cada classe para definir uma nova operação. (GAMMA et al., 1995)

Comportamento relacionado não é espalhado sobre as classes que definem a estrutura do objeto; ele é localizado em um *Visitor*. Conjuntos de comportamentos não-relacionados são particionados nas subclasses de seus próprios *Visitors*. (GAMMA et al., 1995)

### 3.4 *Design Patterns* Arquiteturais

Além dos *design patterns* criados pela GOF, surgiram outros *patterns* com o objetivo de melhorar a estrutura organizacional utilizada no desenvolvimento de *software*. Segundo Fowler et al. (2002), sistemas com uma alta complexidade são comumente divididos em camadas lógicas, onde os principais subsistemas do *software* poderão estar em camadas distintas, onde cada camada superior se comunicará com a inferior. Estas camadas possuem funções variadas e, graças ao conceito por trás dos *design patterns*, as camadas comumente utilizadas pelos desenvolvedores puderam ser catalogadas para se tornarem *design patterns* reutilizáveis.

#### 3.4.1 *Model-View-Controller*

O MVC foi planejado pela primeira vez em meados da década de 70 por Trygve Reenskaug, destinado à linguagem de programação Smalltalk. Desde então, este *pattern* evoluiu e diversas outras implementações surgiram. Apesar de se ter muito debate sobre o MVC e suas evoluções, seu principal objetivo continua sendo o de separar o código da *interface* com o usuário em três áreas separadas. Estas três áreas que o MVC define são: *Model*, *View* e *Controller*, ilustradas na Figura 3.2. (POPE, 2009)

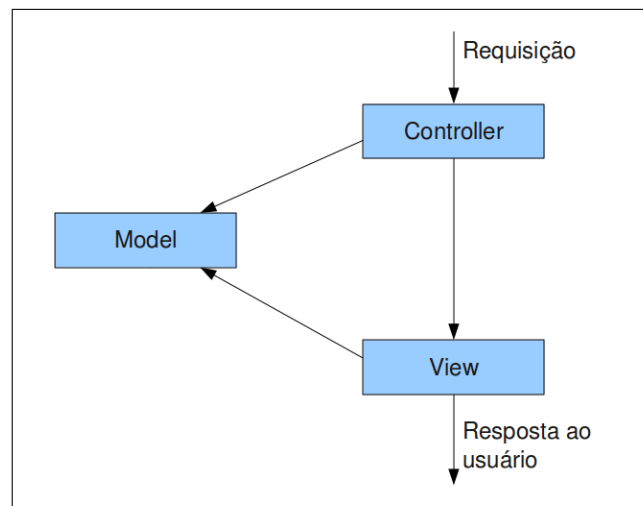


Figura 3.2: MVC Clássico

Melo e Nascimento (2007, p. 231) Adaptada.

O *Model* é a camada que contém a lógica da aplicação e a parte de manipulação dos dados. Esta camada não tem a responsabilidade de processar, por exemplo, requisições HTTP ou lidar com variáveis passadas por requisições deste tipo. (MELO; NASCI-

MENTO, 2007)

A *View* tem a responsabilidade de gerenciar os aspectos de visualização. Ela refletirá os dados de uma classe *Model* e irá formatá-los em uma página *web*, em uma tela de um sistema *desktop*, em *Extensible Markup Language* (XML) ou qualquer outro tipo de apresentação de dados. (MELO; NASCIMENTO, 2007)

E o *Controller* irá fazer o controle da aplicação, onde ele irá manipular o fluxo entre um recurso e outro. Além disso, ele também será o intermediador entre a *View* e o *Model*, já que ambos não podem se comunicar diretamente. (MELO; NASCIMENTO, 2007)

Os principais benefícios de se separar a aplicação desta forma são: simplicidade na adição, edição e exclusão de *interfaces* com o usuário; possuir múltiplas *Views* para apresentar o mesmo dado; facilidade na modificação do controle lógico e ajudar o desenvolvedor a evitar repetição de código. (POPE, 2009)

### 3.4.2 *Data Mapper*

O principal conceito do *Data Mapper* é o de prover uma camada de mapeadores que movem dados entre objetos e um banco de dados, enquanto mantém eles independentes entre si e, também, do próprio mapeador. A Figura 3.3 demonstra por meio de diagramas este conceito. (FOWLER et al., 2002)

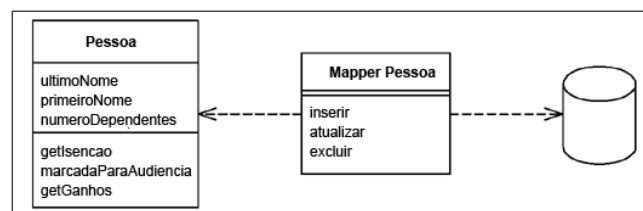


Figura 3.3: *Data Mapper*

Fowler et al. (2002, p. 165) Adaptada.

Objetos e bancos de dados relacionais possuem mecanismos diferentes de se estruturar dados. Muitas partes de um objeto, como coleções e herança, não estão presentes em bancos de dados relacionais. Quando se constrói um modelo de objeto com muita lógica de negócio, é de grande valia utilizar estes mecanismos para organizar melhor os dados e o comportamento que vai com ele. Fazer isso resulta em esquemas variantes, ou seja, o esquema de objeto e o esquema relacional não correspondem. (FOWLER et al., 2002)

O *Data Mapper* é uma camada de *software* que separa objetos em memória e um

banco de dados. Sua responsabilidade é transferir dados entre estes dois e também isolá-los de cada um. Com o *Data Mapper* os objetos em memória: não precisarão saber que existe um banco de dados presente; não precisarão de código SQL; e, certamente, não terão conhecimento do esquema do banco de dados. E, além disso, o *Data Mapper* em si é desconhecido para a camada de domínio. (FOWLER et al., 2002)

### 3.4.3 *Table Data Gateway*

O principal objetivo de um *Table Data Gateway* é o de servir como um ponto de entrada para uma tabela do banco de dados. Uma instância de um *Table Data Gateway* irá manusear todos os registros de uma tabela. (FOWLER et al., 2002)

Misturar instruções SQL à lógica de uma aplicação pode causar diversos problemas. Muitos desenvolvedores podem não dominar SQL, e outros que a dominam podem não escrevê-la corretamente. Administradores de banco de dados precisam ser capazes de encontrar facilmente a parte que lida com SQL da aplicação, para que eles possam melhorar e escrever novos códigos para a comunicação com o banco de dados. (FOWLER et al., 2002)

Um *Table Data Gateway* possui todo SQL utilizado para se efetuar as operações de um banco de dados de uma tabela, como, por exemplo: seleções de dados, inserções, atualizações e exclusões. Este conceito é ilustrado na Figura 3.4. Além destas operações, este tipo de objeto agrega todas as demais operações que interagem com o banco de dados. (FOWLER et al., 2002)

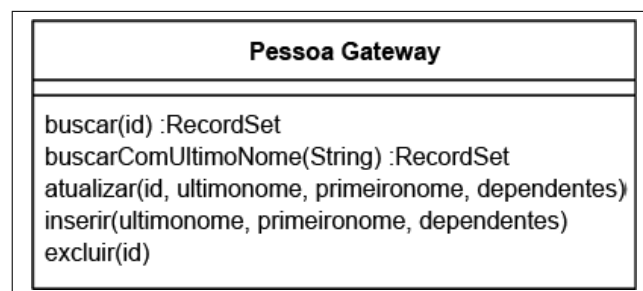


Figura 3.4: *Table Data Gateway*

Fowler et al. (2002, p. 144) Adaptada.

## 4 *PHP*

PHP é um acrônimo recursivo para *Hypertext Preprocessor* (Pré-processador de Hipertexto). É uma linguagem de *script* de código aberto, que tem como objetivo primário a geração de código dinâmico para páginas da *internet*. Isto significa que é possível escrever *Hypertext Markup Language* (HTML) com código PHP embutido para gerar conteúdo dinâmico. O código-fonte de *scripts* PHP não pode ser visto pelo internauta, ele apenas terá acesso ao HTML resultante dos *scripts*. (MELO; NASCIMENTO, 2007)

### 4.1 Histórico

O PHP foi criado no outono de 1994 por Rasmus Lerdorf. Inicialmente a linguagem era formada por um conjunto de scripts voltados à criação de páginas dinâmicas que Rasmus utilizava para monitorar o acesso ao seu currículo na *internet*. Após o crescimento das funcionalidades da linguagem, Rasmus escreveu uma implementação em C, a qual permitia às pessoas desenvolverem de forma muito mais simples suas aplicações *web*. Em 1995 Rasmus nomeou essa versão de *Personal Home Pages/Forms Interpreter* (PHP/FI) e disponibilizou seu código na *web*, para compartilhar com outras pessoas, bem como receber ajuda e correções de *bugs*. (DALL’OGLIO, 2007)

Em 1997 a segunda versão do PHP/FI (PHP/FI 2.0) obteve apoio e reconhecimento de milhares de usuários ao redor do mundo. Aproximadamente 50.000 domínios reportavam sua instalação e uso, construindo assim uma base de 1% dos domínios da *internet*. Esta versão ganhou contribuições de código de milhares de pessoas e foi rapidamente substituída pelas *releases* alfas do PHP 3.0. (MELO; NASCIMENTO, 2007)

Andi Gutmans e Zeev Suraski descobriram que o PHP/FI 2.0 era uma linguagem versátil e que poderia ser utilizada para seus projetos acadêmicos de comércio eletrônico. Em um esforço conjunto a partir da base de usuários PHP/FI existentes, Andi, Rasmus e Zeev decidiram se unir e anunciaram em Junho de 1998 o PHP 3.0 como uma versão



oficial e sucessora do PHP/FI 2.0, que foi descontinuado pelos desenvolvedores. Nesta versão PHP passou a ser um acrônimo para *Hypertext Preprocessor*. Entre os principais recursos desta versão destacam-se: suporte a diversos bancos de dados, protocolos e APIs, suporte à Orientação Objetos (bastante limitado) e uma grande extensibilidade. Nesta versão o PHP estava presente em aproximadamente 10% dos servidores *web* da *internet*. (MELO; NASCIMENTO, 2007)

No inverno de 1998, após o lançamento do PHP 3, Zeev e Andi começaram e trabalhar em uma reescrita do núcleo do PHP, tendo em vista melhorar sua performance e modularidade em aplicações complexas. Para tanto, batizaram este núcleo de *Zend Engine* (Mecanismo Zend), onde Zend era uma mistura entre os nomes Zeev + Andi. O PHP 4, baseado neste mecanismo, foi lançado oficialmente em Maio de 2000, trazendo muitas melhorias e recursos novos, como seções, suporte a diversos servidores *web*, além da abstração de sua API, permitindo inclusive ser utilizado como linguagem para *shell script*. Nesse momento, o PHP já aparecia em cerca de 20% dos domínios da *internet*. (DALL’OGLIO, 2007)

A versão 5 do PHP foi marcada pela quebra de paradigmas da linguagem, pois ela ganhou suporte a Orientação a Objetos de forma muito mais consistente. Esta versão é baseada na *Zend Engine 2*, e foi lançada oficialmente em Junho de 2004. Ela trouxe como novidades o suporte melhorado da manipulação de arquivos XML, manipulação de *webservices Simple Object Access Protocol* (SOAP) e *Representational State Transfer* (REST), suporte melhorado ao MySQL via extensão MySQLi, novas bibliotecas SQLite, Tidy, aperfeiçoamento da integração com a linguagem Perl, melhorias no gerenciamento de memória e fim do suporte ao sistema operacional *Windows 95*. (MELO; NASCIMENTO, 2007)

## 4.2 *Frameworks*

Segundo Melo e Nascimento (2007) um *framework* é um tipo especial de aplicativo, que oferece um conjunto básico de ferramentas e subsistemas que automatizam grande parte dos serviços que se necessita implementar em sistemas para usuários finais. Alguns exemplos práticos são: cadastro de clientes, site de notícias, gestão de estoque e etc.

A utilização de um *framework* é baseada na idéia de que, independente do sistema final do usuário, sempre existirá uma série de componentes padrões, como: controle de usuários, sessões de páginas, *logging* de ações efetuadas, *template engines*, mecanismos de

acesso ao banco de dados e etc. Ao invés de se criar as aplicações do zero, reinventado a roda todas as vezes, pode-se optar pela adoção de um *framework*, de modo que este ofereça uma infra-estrutura completa para sustentação de um sistema. A partir desta premissa, pode-se dedicar o tempo de trabalho apenas no desenvolvimento do núcleo da aplicação, como, por exemplo, as regras de negócio. (MELO; NASCIMENTO, 2007)

Com certeza desta forma tem-se uma enorme economia de tempo e trabalho para o desenvolvimento. Esta é, aliás, a filosofia *Don't Repeat Yourself* (DRY), que prega que elementos estruturais não devem ser reescritos a cada nova aplicação. Além disso, a adoção de um *framework* faz com que o programador adote um estilo mais legível e claro para manter seu código, uma vez que ele deve ser compatível com o modelo do *framework* escolhido. (MELO; NASCIMENTO, 2007)

Diversas linguagens de programação contam com uma série de *frameworks*, construídos e disponibilizados gratuitamente na *internet*. O mesmo ocorre com o PHP, onde entre os tipos de *frameworks* mais utilizados encontram-se: persistência de dados, *template engines*, *frameworks* MVC e de integração com *Asynchronous Javascript And XML* (AJAX).

#### 4.2.1 Zend *Framework*

O Zend *Framework* é um *framework* de código aberto para desenvolvimento de aplicações e serviços *web* com PHP5. Ele foi implementado utilizando código 100% orientado a objetos. A estrutura de componentes do Zend *Framework* é de certo modo única; cada componente é projetado com poucas dependências de outros componentes. Esta arquitetura fracamente acoplada permite aos desenvolvedores utilizar componentes de maneira individual. A companhia Zend é a principal patrocinadora deste *framework*, porém diversas outras companhias contribuem com componentes ou recursos significantes, como, por exemplo, Google, Microsoft e etc. (ZEND, 2009)

A arquitetura do Zend *Framework* possui componentes que abrangem 80% das funcionalidades necessárias a projetos de *software* e ela possui como principal filosofia a possibilidade de se criar componentes que sejam fáceis de utilizar para que se possa chegar aos 20% restantes das funcionalidades de um *software*, geralmente estes sendo os requisitos de negócio de um determinado projeto em questão. Tendo em foco as necessidades mais comuns destes projetos e mantendo o espírito da programação PHP, este *framework* possui uma baixa curva de aprendizagem, o que também reduz os custos de treinamentos. (ZEND, 2009)

De acordo com a Zend (2009), os principais recursos para desenvolvimento *web* fornecidos pelo *framework* são:

- Suporte a AJAX através de *JavaScript Object Notation* (JSON);
- Versão nativa para PHP do mecanismo de busca Lucene;
- Suportar os formatos de dados de sindicacões como, por exemplo *Really Simple Syndication* (RSS), e manipulá-los;
- *Webservices*, consumir e distribuir serviços *web*;
- Biblioteca de classes de alta qualidade escritas em PHP 5, visando as melhores práticas como *design patterns*, testes unitários e etc.

Entre os diversos componentes que o *Zend Framework* possui, existem os que aplicam o MVC, visando a independência de trabalho entre os *web designers* e os programadores; os componentes para bancos de dados, que utilizam abstração de *Structured Query Language* (SQL) e escondem detalhes diversos sobre os Sistemas Gerenciadores de Bancos de Dados (SGBDs); componentes de internacionalização e localização, que prometem facilitar o suporte a múltiplos-idiomas de uma aplicação; e diversos outros componentes como Autenticação, *Web Services*, *Email*, Buscas e também os componentes do núcleo do *framework*. (ZEND, 2009)

#### 4.2.2 *CakePHP*

O *CakePHP* é um *framework* gratuito e de código-aberto para desenvolvimento ágil de aplicações em PHP. Ele fornece toda a estrutura funcional para que programadores possam criar aplicações *web*. O principal objetivo do *CakePHP* é fornecer uma maneira de se trabalhar padronizadamente e com agilidade no desenvolvimento - sem perda de flexibilidade. Com esta premissa, os desenvolvedores podem se focar somente na lógica específica das aplicações. (CAKESOFTWAREFOUNDATION, 2009)

Alguns dos principais recursos, segundo CakeSoftwareFoundation (2009), são:

- Compatibilidade com as versões 4 e 5 do PHP;
- Arquitetura MVC;

- *Scaffolding* (operações básicas de qualquer aplicação em tempo de execução) de Aplicações;
- Gerador de Código;
- Endereços *web* amigáveis;
- Validação nativa.

O *CakePHP* possui uma arquitetura já definida para se trabalhar o que obriga o desenvolvedor a seguir convenções de programação. Entre estas convenções estão a nomenclatura e estrutura de pastas dos *Controllers*, *Models* e *Views* para que suas requisições possam funcionar da maneira correta.

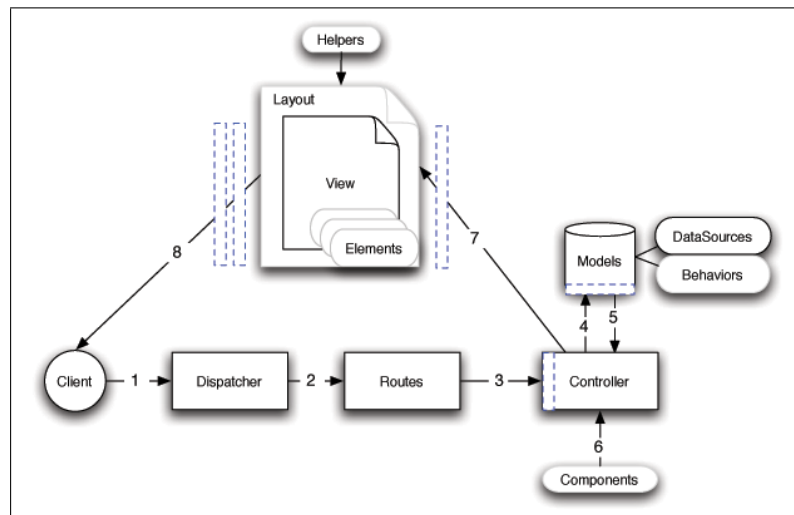


Figura 4.1: Requisição CakePHP.  
(CAKESOFTWAREFOUNDATION, 2009)

Uma requisição com o *CakePHP* é ilustrada na Figura 4.1, onde o usuário faz uma requisição, o *dispatcher* passa essa requisição para as rotas, as rotas lêem a *Uniform Resource Locator* (URL) e extraem os parâmetros dela (*controller*, ação e argumentos utilizados em uma ação), o *Controller* pode utilizar o *Model* para acessar dados do banco de dados, o *Controller* pode utilizar componentes para refinar os dados ou realizar outras operações, o *Controller* passa a resposta para a *View* adequada que é apresentada no navegador do usuário. (CAKESOFTWAREFOUNDATION, 2009)

### 4.2.3 Prado

A inspiração original do Prado veio da *Apache Tapestry*. Durante o projeto e a construção do *framework*, foram copiadas diversas idéias do *Borland Delphi* e do *Microsoft ASP.NET*. A primeira versão do Prado foi lançada em Junho de 2004, escrita em PHP 4. Para concorrer no concurso de codificação PHP 5 da *Zend*, o Prado foi reescrito utilizando os recursos avançados de orientação a objetos desta nova versão da linguagem, onde a versão reescrita do *framework* venceu o concurso. O Prado foi hospedado na *SourceForge* como um projeto de código aberto em Agosto de 2004. Após um grande suporte do time de desenvolvimento e dos usuários do Prado, em 2005 foi lançada a versão 2.0 do projeto. Em Maio de 2005 os desenvolvedores decidiram reescrever completamente o *framework*, para incluir algumas funcionalidades interessantes do *Microsoft ASP.NET 2.0* e para a correção de *bugs*, resultando na versão 3.0 lançada em Abril de 2006. (XUE; ZHUO, 2009)

A arquitetura do Prado é baseada em componentes e eventos e um de seus principais objetivos é habilitar ao máximo a reusabilidade no desenvolvimento *web*. Não somente reutilizar os códigos próprios do desenvolvedor e sim códigos de diversas pessoas. O conceito de componentes é utilizado para este propósito. O Prado utiliza-se do paradigma de programação orientada a eventos para facilitar a interação com os componentes. (XUE; ZHUO, 2009)

Segundo Xue e Zhuo (2009) alguns dos principais recursos do Prado são:

- Reusabilidade;
- Programação orientada a eventos;
- Integração da equipe de programadores;
- Controles *Web* Poderosos;
- Suporte a AJAX.

### 4.2.4 Code Igniter

O *Code Igniter* é um conjunto de ferramentas para desenvolvimento de aplicações *web* utilizando PHP. Seu objetivo é permitir um aumento na agilidade de desenvolvimento de projetos, fornecendo uma biblioteca de componentes para as tarefas comumente

necessárias, e uma estrutura lógica para acessar estes componentes. Com ele o desenvolvedor pode focar no negócio exigido pelo projeto, pois o *framework* minimiza a quantidade de código exigida para uma determinada tarefa. (CODEIGNITER, 2009)

As licenças do *Code Igniter* são de código aberto e bastante flexíveis, pois ele é distribuído sob licenças Apache/BSD, o que permite utilizá-lo para qualquer fim. Ele é compatível com o PHP4 e 5, apesar de não tirar proveito de todos os recursos avançados de orientação a objetos desta última versão. Ele é um *framework* muito leve e seu núcleo não exige muitas bibliotecas. (CODEIGNITER, 2009)

Entre os principais recursos do *Code Igniter*, destacam-se: arquitetura baseada no *pattern* MVC, geração de endereços *web* amigáveis, classes de abstração com o banco de dados, validação de dados e formulários, alta extensibilidade, segurança e filtragem de *Cross-site Scripting* (XSS), entre outras. (CODEIGNITER, 2009)

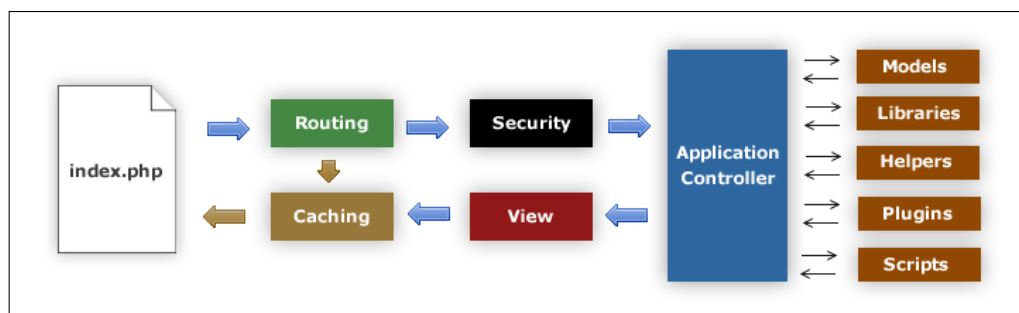


Figura 4.2: Requisição do *Code Igniter*  
CODEIGNITER (2009).

A Figura 4.2 ilustra o fluxo de uma requisição com o *Code Igniter*. Segundo CODEIGNITER (2009) o *framework* processa as requisições da seguinte forma:

- O *Front Controller* (index.php) inicia os recursos necessários para executar o *framework*;
- A requisição HTTP é analisada pelo *Router* para determinar o que deverá ser feito;
- Se existe um arquivo em *cache*, ele é apresentado diretamente ao *browser*, evitando o resto de processamento da requisição;
- Antes do *Controller* ser executado, a requisição HTTP e qualquer dado submetido pelo usuário é filtrado por questões de segurança;
- O *Controller* carrega o *Model*, bibliotecas do núcleo do *framework*, *plugins*, *helpers*, e demais recursos necessários da requisição;

- A *View* resultante é renderizada e enviada ao *browser* para poder ser visualizada. Se o recurso de *caching* está habilitado ela é armazenada para que em requisições posteriores o *framework* possa parar o fluxo na terceira etapa deste processamento.

## 5 *Ambiente Experimental*

O objetivo deste capítulo é descrever sucintamente o ambiente em que foram feitos os testes, os experimentos, as implementações e demais etapas envolvidas no desenvolvimento da arquitetura e da aplicação de gestão de bibliotecas. O ambiente experimental descrito vai desde ferramentas para criação da documentação da aplicação até o ambiente de servidor onde é armazenado o banco de dados e demais tecnologias necessárias para o correto funcionamento da aplicação.

### 5.1 Tecnologias Envolvidas

#### 5.1.1 UML

A UML é um modelo descritivo de objetos largamente utilizado na Engenharia de *Software*. Ela é uma linguagem visual e constitui-se de vários elementos gráficos que permitem construir diagramas para representar as várias visões de um sistema. (MELO; NASCIMENTO, 2007)

Ela foi utilizada na construção da documentação das implementações realizadas. Para desenvolver a UML foi utilizada uma ferramenta visual, Astah\*, descrita na parte de aplicativos.

#### 5.1.2 PHP

O PHP é uma linguagem de *script* amplamente utilizada que é destinada ao desenvolvimento *web* e pode ser misturada com HTML. Ele foi criado por Rasmus Lerdorf e hoje é mantido pela companhia Zend e, também, pela comunidade. O que mais chama a atenção no PHP é que ele é extremamente simples para iniciantes, porém pode fornecer muitos recursos avançados para programadores profissionais. (PHP, 2009)

Como este projeto exige um forte suporte ao paradigma de programação orientada a



objetos, é necessário utilizar os pacotes mais recentes da linguagem, a partir da versão 5. A versão utilizada é a 5.2.10.

### 5.1.3 Apache HTTP

O projeto Apache HTTP é um esforço de se desenvolver um servidor *Hypertext Transfer Protocol* (HTTP) de código aberto para sistemas operacionais modernos, como UNIX e *Windows NT*. O objetivo do projeto é prover um servidor que seja seguro, eficiente e extensível, e que forneça serviços HTTP em sincronia com o padrão atual deste protocolo. (APACHE, 2009)

O servidor Apache foi instalado na máquina servidora da aplicação, a versão escolhida é a 2.2.12.

### 5.1.4 HTML

HTML é um acrônimo de *HyperText Markup Language*, traduzido como Linguagem de Marcação para HiperTexto. É uma linguagem destinada à escrita de documentos que possam ser lidos por *softwares* genericamente chamados de agentes de usuário, como, por exemplo, um navegador ou um leitor de tela. CSS é a abreviação de *Cascading Style Sheet*, que significa Folhas de Estilo em Cascata, e é um mecanismo simples para adicionar estilização aos documentos *web* escritos em HTML. (SILVA, 2008)

A *interface* com o usuário da aplicação foi construída utilizando HTML e CSS, e as premissas da parte visual é focar na simplicidade e usabilidade do sistema.

### 5.1.5 MySQL

O MySQL é o *software* de banco de dados de código aberto mais popular do mundo, com milhões de distribuições já feitas pela *internet*. Entre seus principais recursos, destacam-se: *performance*, confiabilidade e facilidade de uso ele se tornou a escolha preferida para sistemas que rodam na *web*. Atualmente ele pertence à *Sun Microsystems*. (MYSQL, 2009)

A versão do MySQL utilizada no projeto é a 5.1.37.

## 5.2 Padrões Envolvidos

### 5.2.1 Programação Orientada a Objetos

A orientação a objetos é um paradigma de programação, da mesma forma como a programação estruturada. A principal diferença são nos conceitos que envolvem a orientação a objetos. Segundo Dall'Oglio (2007), os principais conceitos da orientação a objetos são:

- **Classes:** É uma estrutura que define um tipo de dados, que pode conter atributos (variáveis) e funções (métodos) para manipular estes atributos;
- **Objetos:** Um objeto contém exatamente a mesma estrutura e as propriedades de uma classe, no entanto sua estrutura é dinâmica, seus atributos podem mudar de valor durante a execução do programa e pode-se declarar diversos objetos oriundos da mesma classe;
- **Atributos:** Representam as características de um objeto e definem a estrutura do mesmo;
- **Métodos:** Ações ou serviços que um objeto pode executar;
- **Modificadores de Acesso:** Referem-se aos modificadores *Public*, *Private* e *Protected*, e definem o nível de encapsulamento de um determinado atributo ou método.

Devido aos *design patterns* serem direcionados e desenvolvidos para projetos que utilizam orientação a objetos, este paradigma foi obrigatório no experimento deste projeto.

### 5.2.2 *Design Patterns*

Com o intuito de se prover uma arquitetura de *software* para desenvolver diversos tipos de aplicação, de uma forma mais padronizada, é necessário se aplicar os *design patterns*, para resolver problemas com soluções reutilizáveis e para definir formas mais organizadas de se trabalhar. Na implementação foram aplicados diversos *patterns*, abaixo encontra-se uma lista deles, junto com uma breve definição retirada da fundamentação teórica deste trabalho:

- ***Singleton:*** Garante que apenas uma instância de determinada classe seja permitida na aplicação;

- **Factory:** Centraliza a instanciação de determinados objetos da aplicação;
- **Facade:** Esconde de camadas superiores longos trechos de códigos de subsistemas;
- **Observer:** Toma ações ao ser notificado de que algum evento ocorreu em classes observáveis;
- **Data Mapper:** Faz o mapeamento entre os dados do banco de dados com os objetos da aplicação;
- **Table Data Gateway:** Age como uma porta de entrada para uma tabela do banco de dados. Irá lidar com todas as linhas da tabela;
- **MVC:** Divide a aplicação em três camadas: *Model* que possuirá a lógica de negócio, *View* que apresentará os dados e *Controller* que interligará as outras duas camadas.

## 5.3 Estrutura Física

### 5.3.1 Ambiente Físico

O ambiente onde foram feitos os experimentos fica a cargo de uma máquina doméstica convencional para armazenar a aplicação e de uma máquina com um navegador de internet para acessar a aplicação. A rede utilizada possui apenas um hub para permitir acesso entre os computadores, e a velocidade da conexão é *Fast Ethernet* 100.

### 5.3.2 Configurações de *Hardware*

As máquinas que serão utilizadas nos experimentos possuem as seguintes configurações:

- **Servidor - Máquina Doméstica**
  - CPU Sempron 3800+;
  - 512 MB de memória DDR 400MHz;
  - HD de 80 GB PATA.
- **Cliente - Notebook SempToshiba IS1525**
  - CPU Pentium Dual Core T2130 1.86GHz;

- 2 GB de memória DDR2 667MHz;
- HD de 160 GB SATA.

## 5.4 Estrutura Lógica

### 5.4.1 Sistema Operacional

O sistema operacional que será utilizado em ambas as máquinas é o GNU/Linux Ubuntu 9.10 Karmic Koala, com todas as atualizações disponíveis já instaladas. O kernel utilizado é o 2.6.31-14-generic. Os pacotes utilizados são:

- apache2 - Versão 2.2.12;
- php5 - Versão 5.2.10;
- pdo-mysql - Versão 5.1.37;
- mysql-server-5.1 - Versão 5.1.37.

### 5.4.2 Aplicativos

#### 5.4.2.1 Astah\* Community

Astah\* é um editor UML que fornece diversos recursos, entre eles Mapas Mentais, diagramas de classes, casos de uso e sequência entre outros. Todos os diagramas criados no Astah\* são consistentemente guardados em um modelo, o que pode dar mais eficiência à comunicação entre membros da equipe de desenvolvimento. O Astah\* possui as versões: *Community*, *UML*, *Professional* e *Share*, onde a *Community* e a *Share* são gratuitas, e a *UML* e *Professional* pagas. (CHANGEVISION, 2009)

O Astah\* *Community* foi escolhido principalmente pela interoperabilidade, pelos recursos oferecidos e por ser um *software* gratuito, a versão utilizada é a 6.0, por ser a última versão estável do *software*.

#### 5.4.2.2 Eclipse

O Eclipse é um ambiente de desenvolvimento integrado (*Integrated Development Environment*) focado principalmente no desenvolvimento de aplicações Java, que pode ser facilmente estendido para oferecer suporte a outras linguagens, como o PHP, a partir

de *plugins*. O projeto Eclipse foi criado originalmente pela IBM em Novembro de 2001, tornando-se independente após a criação da Eclipse *Foundation*, em Janeiro de 2004. O Eclipse é distribuído como *software* livre e é um ambiente desenvolvido na linguagem Java, fornecendo interoperabilidade, necessitando apenas de uma máquina virtual Java para ser executado.

O Eclipse utilizado é um pacote específico para programação de aplicações em PHP, o Eclipse PDT, e a versão do pacote é a 2.1.

#### 5.4.2.3 phpMyAdmin

O phpMyAdmin é uma ferramenta gratuita escrita em PHP com o objetivo de administrar bancos de dados MySQL. Esta ferramenta suporta diversas operações com o MySQL. Sua *interface* com o usuário suporta as operações mais utilizadas, como: administrar bancos de dados, tabelas, campos, relacionamentos, índices, usuários, permissões, etc. Além de existir a possibilidade de se executar diretamente instruções SQL a partir de um mini-terminal *web*. (PHPMYADMIN, 2009)

Ele foi selecionado por ser interoperável e prático, sendo a versão utilizada a 3.2.3.

### 5.4.3 Frameworks

#### 5.4.3.1 Zend Framework

O Zend *Framework* foi desenvolvido com o foco em se ter uma ferramenta de extrema simplicidade e produtividade, que possua os últimos recursos disponíveis da *web* 2.0 e que possua um código bem testado, garantindo também a agilidade nesses testes, onde empresas possam depender deste código sem maiores problemas. (ZEND, 2009)

O *framework* fornece uma biblioteca de componentes de baixo acoplamento que provê 80% das funcionalidades que os desenvolvedores necessitam e que permite aos desenvolvedores a customização dos 20% restantes para atingir os requisitos específicos necessários. Ele também possui os seguintes recursos disponíveis: suporte a AJAX através de JSON, mecanismo Lucene para realizar buscas, sindicâncias de dados, *web services* e utilização de todos os recursos avançados de orientação a objetos da linguagem PHP a partir da versão 5. (ZEND, 2009)

O Zend *Framework* é o núcleo da arquitetura, a partir dele foram aplicados os *design patterns* selecionados e foi construída a aplicação. A versão utilizada é a 1.9.5.

## 6 *Arquitetura Definida*

A arquitetura de *software* definida é uma arquitetura N-camadas, onde são delegadas responsabilidades distintas para cada uma destas camadas. Estas camadas são camadas lógicas, sendo feita esta divisão para facilitar a distribuição dos componentes da aplicação. A Figura 6.1 apresenta as camadas da arquitetura e a relação que elas possuem entre si.

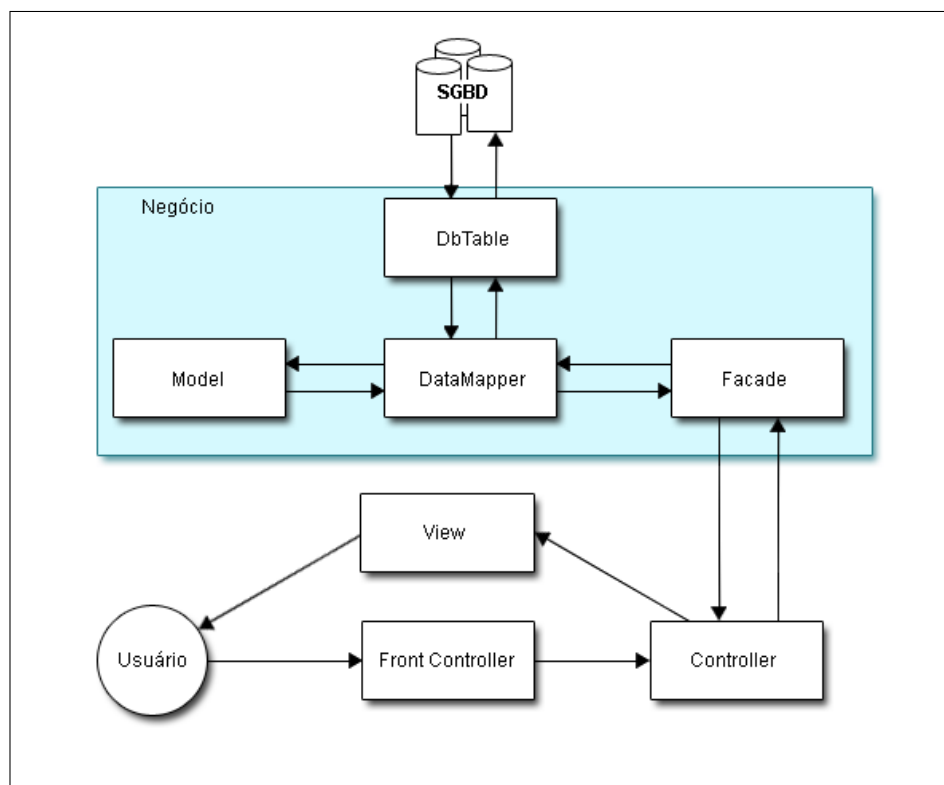


Figura 6.1: Camadas da Arquitetura de *Software*.

### 6.1 Fluxo da Arquitetura

O fluxo definido por esta arquitetura segue o padrão definido pelo Zend *Framework* e, também, possui algumas customizações para a comunicação entre cada camada. A estrutura base do *framework* é baseada no *pattern* MVC, o que divide a aplicação em

*Model*, *View* e *Controller*. Na arquitetura, conforme apresentado na Figura 6.1, ainda existem as camadas: *Facade*, *Data Mapper* e *Table Data Gateway*.

Todo o fluxo inicia-se por uma requisição feita por um usuário, o *framework* definirá qual o *Controller* requerido, este então será responsável por tratar a requisição e, utilizando o *pattern Factory Method*, o *Controller* obtém a *Facade* ligada ao caso de uso a que ele corresponde e então delega para esta camada o processamento da lógica referente a regra de negócio.

A *Facade* poderá utilizar um *Data Mapper* para obter dados do banco de dados, ou para fazer operações a registros do banco. O *Data Mapper* irá utilizar o *Table Data Gateway* para efetuar as operações SQL, que é a linguagem compreendida pelo banco de dados. Ele também poderá mapear os dados vindos do *Table Data Gateway* para objetos *Model*, que representam em forma de objetos as entidades do banco de dados.

Existe ainda a implementação do *Observer* e *Observable*, que fazem parte do *design pattern Observer*. Uma classe *Observable* possuirá métodos para se conectar a *Observers* e para notificar cada um deles. A classe *Observer* irá fazer um *log* das operações notificadas pela *Observable*, gravando este *log* em banco de dados, no formato JSON, para permitir uma consulta posterior.

Após todo o processamento das camadas inferiores ser concluído, o *Controller* irá continuar o fluxo da aplicação, exibindo a *View* para o usuário, que pode conter os dados pegos pelo *Data Mapper*, ou os formulários definidos pelos componentes *Zend\_Form* para obter dados para algum registro, ou mensagens relevantes para informar ao usuário.

O *design pattern Singleton* é implementado por diversos componentes do *Zend Framework*, como, por exemplo, o *Zend\_Auth* que é utilizado na autenticação de usuários. Com este *pattern* é possível manter os objetos durante a aplicação e por apenas um ponto de entrada. Isto garante a consistência deste objeto, sabendo sempre o que esperar dele.

## 7 *Implementação*

Para demonstrar o uso da arquitetura proposta foi implementada uma aplicação de controle de bibliotecas, com alguns casos de uso que poderiam ser utilizados por bibliotecas reais. Este capítulo descreve esta implementação, contendo a documentação do sistema e a aplicação em si.

Para implementar a arquitetura de *software* proposta neste trabalho, foi especificado um sistema de biblioteca para um cliente fictício, este sendo uma biblioteca. Com a automação que sistemas computacionais visam fornecer a clientes e os casos de uso implementados, alguns problemas encontrados em bibliotecas foram solucionados, utilizando uma implementação bem estruturada graças aos *design patterns*.

### 7.1 Documentação

Uma análise sucinta foi utilizada para documentar o que foi desenvolvido no sistema, utilizando diagramas UML.

#### 7.1.1 Casos de Uso

Foi elaborado um diagrama de casos de uso que demonstra cada funcionalidade que um administrador da biblioteca poderá realizar. Este encontra-se na Figura 7.1.



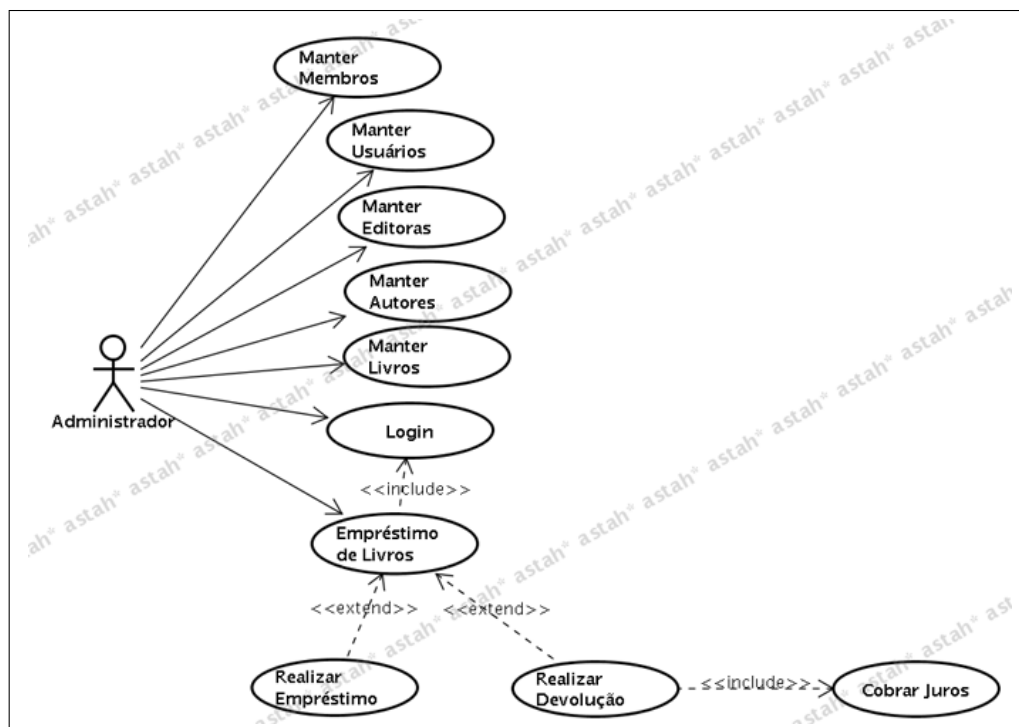


Figura 7.1: Diagrama de Casos de Uso.

Na tabela 7.1 encontra-se a especificação de cada caso de uso, descrevendo o que cada caso de uso representa no escopo das funcionalidades do sistema.

Nome do Caso de Uso	Descrição
Login	Processo de autorização dos usuários do sistema.
Manter Usuários	Cadastro, Alteração, Visualização, Listagem e Exclusão dos usuários aptos a utilizar o sistema.
Manter Membros	Cadastro, Alteração, Visualização, Listagem e Exclusão dos membros da biblioteca.
Manter Editoras	Cadastro, Alteração, Visualização, Listagem e Exclusão de editoras de livros.
Manter Autores	Cadastro, Alteração, Visualização, Listagem e Exclusão de autores de livros.
Manter Livros	Cadastro, Alteração, Visualização, Listagem e Exclusão dos livros que compõem o acervo da biblioteca.
Realizar Empréstimo	Processo de empréstimo de um ou vários livros de um membro da biblioteca.
Realizar Devolução	Faz a devolução de um ou mais livros que foram emprestados a um determinado membro.
Cobrar Juros	Caso a devolução seja feita a uma data acima da prevista de devolução, é cobrado uma taxa de juros ao membro.

Tabela 7.1: Especificação dos casos de uso

### 7.1.2 Diagrama de Classes

Após ter uma especificação de cada funcionalidade que o sistema deve prever, foi elaborado um diagrama de classes. Este diagrama é referente somente as classes que formam o domínio de negócio do sistema, ou seja, objetos que representam entidades utilizadas ao longo do sistema como, por exemplo, Livro ou Empréstimo, e que são específicas da aplicação em questão. As classes de domínio de negócio possuem somente as características do objeto em questão. Este diagrama de classes encontra-se na figura 7.2.

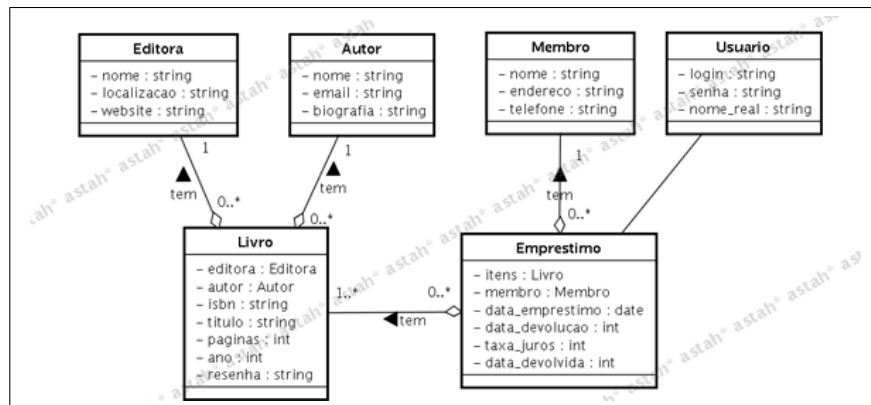


Figura 7.2: Diagrama de Classes.

### 7.1.3 Banco de Dados

Como descrito no capítulo de ambiente experimental o SGBD utilizado foi o MySQL. Para armazenar os dados da aplicação foram definidas as tabelas utilizadas por cada caso de uso, tabelas estas que seguem a 3ª Forma Normal de bancos de dados relacionais. Foi gerado um documento de Modelo de Entidades e Relacionamentos (MER), este modelo é apresentado na figura 7.3.

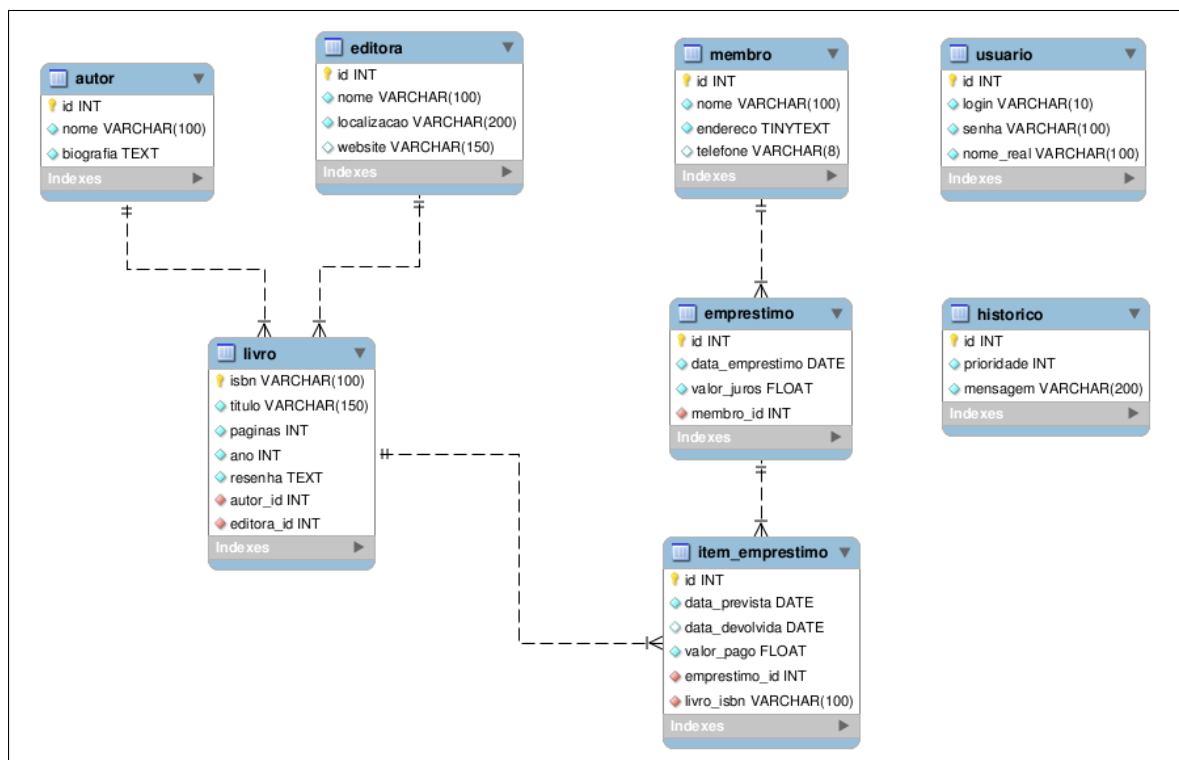


Figura 7.3: Modelo de Entidades e Relacionamentos.

## 7.2 Desenvolvimento da Aplicação

O Zend *Framework* fornece uma estrutura padrão para se organizar cada parte de uma aplicação. Baseando-se nesta convenção, foi estruturada uma hierarquia customizada para se aplicar os *design patterns* da arquitetura. Além dos pacotes de cada camada, ainda existem pacotes para *helpers*, configurações e internacionalização. A Figura 7.4 demonstra a hierarquia de pastas definida para a aplicação.

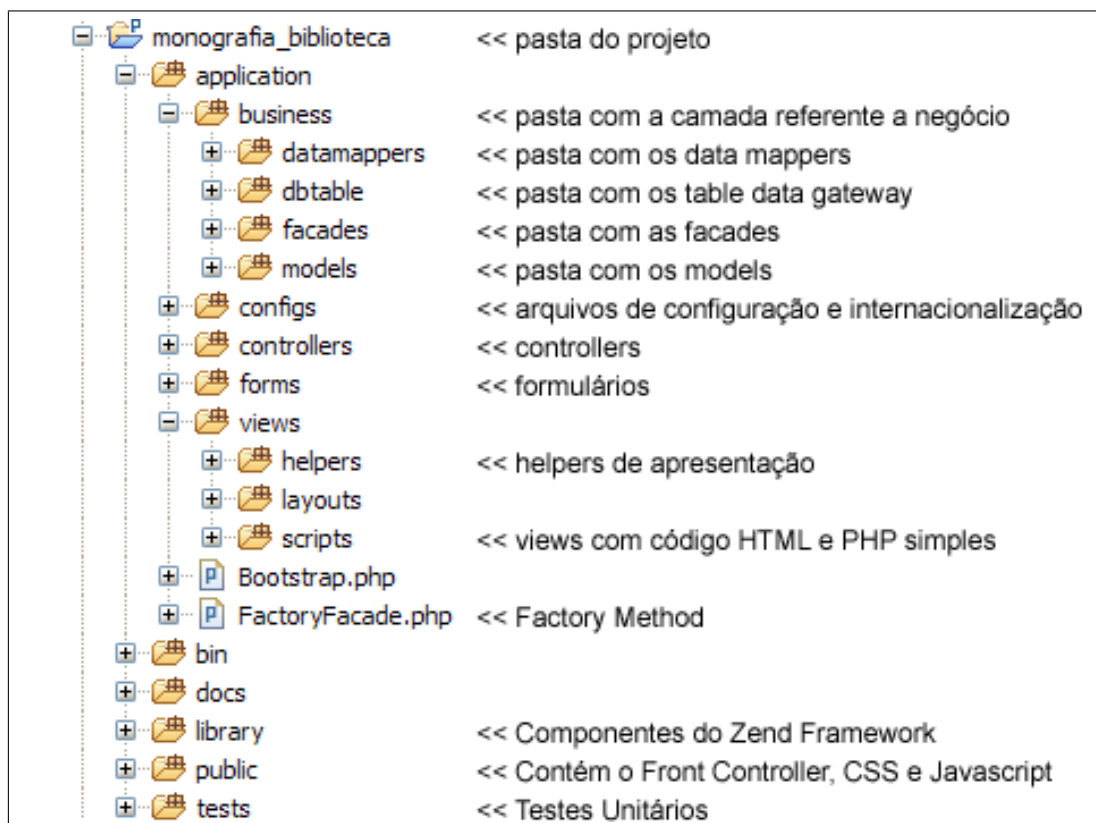


Figura 7.4: Estrutura de Diretórios.

A Listagem 7.1 é o arquivo de configuração utilizado para definir alguns parâmetros de componentes, assim como os dados de acesso a banco de dados. O arquivo é baseado na sintaxe INI utilizada para estruturação de arquivos de configuração no geral.

Listagem 7.1: Arquivo de Configuração

```

1  [production]
2  includePaths.library = APPLICATION_PATH "../library"
3  bootstrap.class = "Bootstrap"
4  resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
5  resources.frontController.defaultControllerName = "emprestimo"
6  # Banco de dados
7  resources.db.adapter = "PDO_MYSQL"
8  resources.db.params.host = "localhost"
9  resources.db.params.username = "root"
10 resources.db.params.password = ""
11 resources.db.params.dbname = "monografia_biblioteca"

```

## 7.2.1 Negócio

A camada de negócio é responsável pelas regras de negócio que a aplicação necessitará assim como acesso a banco de dados, validações e demais operações de mais baixo nível.

### 7.2.1.1 *Facade*

Como já dito, a *Facade* conterà lógica de negócio, escondendo código das camadas superiores. As *Facades* são instanciadas a partir do uso de uma *Factory*, que é apresentada mais adiante. A Listagem 7.2 apresenta o código da *Facade* de Empréstimos que, nesta aplicação, é a que mais possui regras de negócio.

Listagem 7.2: *Facade* de Empréstimo

```

1 class Biblioteca_Business_Facade_Emprestimo {
2     protected $_mapper;
3     protected $_mapperItem;
4
5     public function __construct() {
6         $this->_mapper = new Biblioteca_Business_DataMapper_Emprestimo();
7         $this->_mapperItem = new Biblioteca_Business_DataMapper_ItemEmprestimo();
8     }
9     public function calculaJurosBusiness($id, $data_devolucao) {
10         $item = $this->_mapperItem->get($id);
11         $date_validator = new Zend_Validate_Date();
12         if ($date_validator->isValid($data_devolucao)) {
13             $valor_juros = $item->getEmprestimo()->getValorJuros();
14             $data_prevista = $item->getDataPrevista();
15
16             $timestamp_data_devolucao = mktime(0, 0, 0, substr($data_devolucao, 5, 2),
17                 substr($data_devolucao, 8, 2), substr($data_devolucao, 0, 4));
18             $timestamp_data_prevista = mktime(0, 0, 0, substr($data_prevista, 5, 2),
19                 substr($data_prevista, 8, 2), substr($data_prevista, 0, 4));
20
21             $diferenca = ($timestamp_data_devolucao - $timestamp_data_prevista);
22             if ($diferenca <= 0) {
23                 return 0;
24             }
25             else {
26                 $dias_decorridos = ceil($diferenca / 86400);
27                 return $valor_juros * $dias_decorridos;
28             }
29         }
30         return false;
31     }
32     public function devolveItemBusiness($data, $form) {
33         if ($form->isValid($data)) {
34             $id = $form->getValue('id');
35             $data_devolvida = $form->getValue('data_devolvida');
36             $valor_pago = $this->calculaJurosBusiness($id, $data_devolvida);
37
38             $this->_mapperItem->devolucao($id, $data_devolvida, $valor_pago);
39
40             return true;
41         }
42         else {
43             return false;
44         }
45     }
46     protected function verificaItemEmprestadoBusiness($isbn) {
47         if ($item = $this->_mapperItem->getByIsbn($isbn)) {
48             if (is_null($item->getDataDevolvida())) {
49                 return true;
50             }
51         }
52     }
53 }

```

```

48         }
49         return false;
50     }
51     return false;
52 }
53 }

```

A partir deste código pode-se ver o conceito deste *pattern* em ação, o código de uma *Facade* pode acabar sendo extenso, mas ele fica concentrado em um ponto, onde sua função é realmente ser extenso para que outras camadas acabem ficando com menos código. Esta *Facade* está resumida, pois ainda existem os métodos relativos à adição, edição e exclusão de itens.

### 7.2.1.2 *Factory Method*

O *pattern Factory Method* foi utilizado para abstrair a instanciação de *Facades* pelo *Controller*. O código da *Factory* é mostrado na Listagem 7.3.

Listagem 7.3: *Factory* de *Facades*

```

1  class FactoryFacade {
2      const FACADEEDITORIA = 1;
3      const FACADEAUTOR = 2;
4      const FACADELIVRO = 3;
5      const FACADEEMPRESTIMO = 4;
6      const FACADEUSUARIO = 5;
7      const FACADEMEMBRO = 6;
8      const FACADEAUTH = 7;
9
10     public static function obterFacade($facade) {
11         switch ($facade) {
12             case self::FACADEEDITORIA:
13                 return new Biblioteca_Business_Facade_Editoria();
14             break;
15             case self::FACADEAUTOR:
16                 return new Biblioteca_Business_Facade_Autor();
17             break;
18             case self::FACADELIVRO:
19                 return new Biblioteca_Business_Facade_Livro();
20             break;
21             case self::FACADEEMPRESTIMO:
22                 return new Biblioteca_Business_Facade_Emprestimo();
23             break;
24             case self::FACADEUSUARIO:
25                 return new Biblioteca_Business_Facade_Usuario();
26             break;
27             case self::FACADEMEMBRO:
28                 return new Biblioteca_Business_Facade_Membro();
29             break;
30             case self::FACADEAUTH:
31                 return new Biblioteca_Business_Facade_Auth();
32             break;
33             default:
34                 throw new Exception('Facade invalida!');
35             break;
36         }
37     }
38 }

```

Baseando-se em *flags* definidas na *Factory*, pode-se saber qual a *Facade* que se deseja instanciar e após a instanciação basta retorná-la para as camadas que a exigiram.

### 7.2.1.3 *Data Mapper e Table Data Gateway*

Ambos estão relacionados com a parte de banco de dados da aplicação, onde o *Data Mapper* mapeará dados vindos do *Table Data Gateway* que se comunica diretamente com o banco de dados. O *Table Data Gateway* herda de um componente do *Zend Framework*, o *Zend\_Db\_Table*, todos os métodos referentes a persistência, abstraindo ainda mais a SQL do desenvolvedor.

Nas listagens 7.4 e 7.5 é possível verificar, respectivamente, um *Data Mapper* e um *Table Data Gateway*.

Listagem 7.4: *Data Mapper* de Empréstimo

```

1 class Biblioteca_Business_DataMapper_Emprestimo {
2     protected $_dbTable;
3
4     public function __construct() {
5         $this->setDbTable(new Biblioteca_Business_DbTable_Emprestimo());
6         $this->getDbTable()->attachObserver("BibliotecaUtils_Observer_Observer");
7     }
8     private function setDbTable(Biblioteca_Business_DbTable_Emprestimo $dbtable) {
9         $this->_dbTable = $dbtable;
10    }
11    private function getDbTable() {
12        return $this->_dbTable;
13    }
14    public function add($data_emprestimo, $membro_id, $valor_juros) {
15        $data = array(
16            'data_emprestimo' => $data_emprestimo,
17            'membro_id' => $membro_id,
18            'valor_juros' => $valor_juros
19        );
20        $id = $this->getDbTable()->insert($data);
21        return $id;
22    }
23    public function get($id, $array = false) {
24        $row = $this->getDbTable()->fetchRow('id = ' . (int) $id);
25        if ($row) {
26            $membro = $row->findParentRow('Biblioteca_Business_DbTable_Membro')->toArray();
27            $data = $row->toArray();
28            if ($array) {
29                array_push($data, array('membro' => $membro));
30                return $data;
31            }
32            $model = new Biblioteca_Business_Model_Emprestimo();
33            $model_membro = new Biblioteca_Business_Model_Membro();
34            $model_membro->setId($membro['id'])
35                ->setEndereco($membro['endereco'])
36                ->setNome($membro['nome'])
37                ->setTelefone($membro['telefone']);
38            $model->setId($data['id'])
39                ->setMembro($model_membro)
40                ->setDataEmprestimo($data['data_emprestimo'])
41                ->setValorJuros($data['valor_juros']);
42            return $model;
43        }
44        return false;
45    }
46 }

```

O método `get()` demonstra o conceito do *Data Mapper* onde ele mapeia os dados vindos do banco para um *Model* da aplicação.

Listagem 7.5: *Table Data Gateway* de Empréstimo

```

1 class Biblioteca_Business_DbTable_Emprestimo extends BibliotecaUtils_Observer_Observable {
2     protected $_name = 'emprestimo';
3     protected $_referenceMap = array(
4         'Membro' => array(
5             'columns' => array('membro_id'),
6             'refTableClass' => 'Biblioteca_Business_DbTable_Membro',
7             'refColumns' => 'id'
8         )
9     );
10 }

```

A partir da Listagem 7.5 é possível perceber a abstração de todo código pertinente ao acesso a banco. A classe *Observable*, detalhada posteriormente, herda de `Zend_Db_Table`, e, um *Table Data Gateway* possuirá todos os métodos de um *Observable* e, também, de um `Zend_Db_Table`.

#### 7.2.1.4 *Model*

O *Model* possui apenas os atributos que definem uma entidade do banco e os métodos acessores a estes atributos. A listagem 7.6 demonstra um *Model* utilizado na aplicação.

Listagem 7.6: *Model* da entidade Empréstimo

```

1 class Biblioteca_Business_Model_Emprestimo {
2     private $_id;
3     private $_membro;
4     private $_data_emprestimo;
5     private $_valor_juros;
6
7     public function getId() {
8         return $this->_id;
9     }
10    public function setId($id) {
11        $this->_id = $id;
12        return $this;
13    }
14    public function getMembro() {
15        return $this->_membro;
16    }
17    public function setMembro(Biblioteca_Business_Model_Membro $membro) {
18        $this->_membro = $membro;
19        return $this;
20    }
21    public function getDataEmprestimo() {
22        return $this->_data_emprestimo;
23    }
24    public function setDataEmprestimo($data_emprestimo) {
25        $this->_data_emprestimo = $data_emprestimo;
26        return $this;
27    }
28    public function getValorJuros() {
29        return $this->_valor_juros;
30    }
31    public function setValorJuros($valor_juros) {
32        $this->_valor_juros = $valor_juros;
33        return $this;
34    }
35 }

```



35 }

### 7.2.1.5 Observer

O *Observer* trabalha com o conceito de observadores e observáveis, que na arquitetura são representados, respectivamente, pelas classes: *Observer* e *Observable*. Ambas encontram-se nas Listagens: 7.7 e 7.8.

Listagem 7.7: Classe *Observer*

```

1 class BibliotecaUtils_Observer_Observer {
2     protected $_data;
3     protected $_logger;
4
5     protected function setLogger(Zend_Log $logger) {
6         $this->_logger = $logger;
7     }
8     protected function getLogger() {
9         return $this->_logger;
10    }
11    protected function setData($data) {
12        $this->_data = $data;
13    }
14    protected function getData() {
15        return $this->_data;
16    }
17    protected function initLog() {
18        $db = Zend_Db_Table::getDefaultAdapter();
19        $columns = $this->parseColumns();
20        $writer = new Zend_Log_Writer_Db($db, 'historico', $columns);
21        $this->setLogger(new Zend_Log($writer));
22    }
23    protected function createData($event, $class) {
24        $data = array();
25        $data['operacao'] = ($event == 'insert') ? "Novo" : "Atualizacao";
26        if ($event != 'delete') {
27            foreach ($class->data as $key => $value) {
28                $data[$key] = $value;
29            }
30        }
31        else {
32            $data['operacao'] = "Exclusao";
33        }
34        if ($event != 'insert') {
35            $where = explode('=', $class->where);
36            $data['id'] = trim($where[1]);
37        }
38        $usuario = Zend_Auth::getInstance()->getIdentity();
39        $data['efetuado_por'] = $usuario->id;
40        $this->setData(Zend_Json::encode($data));
41    }
42    protected function parseColumns() {
43        $columns = array('prioridade' => 'priority', 'mensagem' => 'message');
44        return $columns;
45    }
46    protected function logMessage() {
47        $this->getLogger()->info($this->getData());
48    }
49    public function observeTable($event, $class) {
50        $this->createData($event, $class);
51        $this->initLog();
52        $this->logMessage();
53    }
54 }

```

A classe *Observer* utiliza o componente *Zend\_Log* para guardar um *log* das operações em banco de dados, permitindo uma consulta posterior, no caso de necessitar de *backups* para recuperação de dados, ou para manter um histórico dos usuários.

Listagem 7.8: Classe *Observable*

```

1 class BibliotecaUtils_Observer_Observable extends Zend_Db_Table_Abstract {
2     public $data;
3     public $where;
4     protected static $_observers = array();
5
6     public static function attachObserver($class) {
7         if (!is_string($class) || !class_exists($class) || !is_callable(array($class, '
8             observeTable')) {
9             return false;
10        }
11        if (!isset(self::$_observers[$class])) {
12            self::$_observers[$class] = true;
13        }
14        return true;
15    }
16    protected function _notifyObservers($event) {
17        if (!empty(self::$_observers)) {
18            foreach (array_keys(self::$_observers) as $observer) {
19                $obj_observer = new $observer();
20                call_user_func(array($obj_observer, 'observeTable'), $event, $this);
21            }
22        }
23    public function insert($data) {
24        $last_id = parent::insert($data);
25        if ($last_id > 0) {
26            $this->data = $data;
27            $this->data['id'] = $last_id;
28            $this->_notifyObservers("insert");
29        }
30        return $last_id;
31    }
32    public function update($data, $where) {
33        parent::update($data, $where);
34        $this->where = $where;
35        $this->data = $data;
36        $this->_notifyObservers("update");
37    }
38    public function delete($where) {
39        $this->where = $where;
40        $last_id = parent::delete($this->where);
41        if ($last_id > 0) {
42            $this->_notifyObservers("delete");
43        }
44        return $last_id;
45    }
46 }

```

As classes observáveis são classes *Table Data Gateway*. A diferença é que a classe *Observable* utiliza o polimorfismo para sobrescrever o comportamento padrão dos métodos de persistência, para incluir a notificação de um determinado evento ao observador.

## 7.2.2 Camada de Mais Alto Nível

A camada de negócio é a mais complexa e que necessitou de mais *patterns* para abstrair as partes que compõem uma aplicação. A camada de mais alto nível é composta

pelo *Controller* e pela *View*, e possui código muito mais simplificado.

Na listagem 7.9, é exibido o código de um *Controller*, onde ele é baseado em *actions* para determinar qual operação está sendo requisitada pelo usuário.

Listagem 7.9: *Controller* de Empréstimo

```

1  class EmpréstimoController extends Zend_Controller_Action {
2      public function init() {
3          if (!Zend_Auth::getInstance()->hasIdentity()) {
4              $this->_redirect('/auth');
5          }
6          $this->facade = FactoryFacade::obterFacade(FactoryFacade::FACADE_EMPRESTIMO);
7      }
8      public function indexAction() {
9          $this->view->headTitle('Empréstimos', 'PREPEND');
10         $this->view->emprestimos = $this->facade->listBusiness();
11     }
12     public function adicionarAction() {
13         $this->view->headTitle('Novo Empréstimo', 'PREPEND');
14         $form = new Biblioteca_Form_Empréstimo();
15         $this->view->form = $form;
16         if ($this->getRequest()->isPost()) {
17             $data = $this->getRequest()->getPost();
18             if ($id = $this->facade->addBusiness($data, $form)) {
19                 $this->_helper->FlashMessenger('Empréstimo adicionado, insira agora os
20                     itens deste empréstimo.');
```

A *View* ficou, com código HTML e apenas alguns trechos de código PHP. Como toda a lógica está definida em camadas mais baixas, a apresentação com o usuário exige apenas funções PHP simples, como, por exemplo, laços para percorrer dados e métodos para imprimir estes dados. A listagem 7.10 refere-se a *View* de listagem de empréstimos e ilustra melhor o que é utilizado nas *Views*.

Listagem 7.10: *View* de Listagem de Empréstimos

```

1 <h2>Listando Empréstimos</h2>
2 <p>
3     <a href="php echo $this-&gt;url(array('action' =&gt; 'adicionar')); ?" class="button_top">Novo
        Empréstimo</a>
4 </p>
5 <table class="grid">
6     <thead>
7         <tr>
8             <th scope="col">Membro</th>
9             <th scope="col">Data do Empréstimo</th>
10            <th scope="col">Valor do Juros</th>
11            <th scope="col">Excluir</th>
12        </tr>
13    </thead>
14    <tbody>
15        <?php if ( sizeof($this->emprestimos) == 0 ): ?>
16            <tr>
17                <td colspan="4">Nenhum registro encontrado.</td>
18            </tr>
19        <?php else: ?>
20        <?php foreach ($this->emprestimos as $emprestimo): ?>
21            <tr>
22                <td><a href="php echo $this-&gt;url(array('action' =&gt; 'adicionaritens', 'id' =&gt;
                    $emprestimo-&gt;getId())); ?"><?php echo $emprestimo->getMembro()->getNome()
                    ; ?></a></td>
23                <td><?php echo $emprestimo->getDataEmprestimo(); ?></td>
24                <td>R$<?php echo $emprestimo->getValorJuros(); ?></td>
25                <td><a href="php echo $this-&gt;url(array('action' =&gt; 'deletar', 'id' =&gt;
                    $emprestimo-&gt;getId())); ?" onclick="return deletar(this.href);">Excluir</
                    a></td>
26            </tr>
27        <?php endforeach; ?>
28        <?php endif; ?>
29    </tbody>
30 </table>

```

O componente `Zend_Form` abstraiu a parte de criação de formulários e de validação. Com o `Zend_Form` basta criar uma classe que herda deste componente e definir os elementos do formulário, junto com as validações para cada elemento. O código do formulário de cadastro de um novo empréstimo é apresentado na listagem 7.11.

Listagem 7.11: Formulário de Empréstimo

```

1 class Biblioteca_Form_Emprestimo extends Zend_Form {
2     private $editar = false;
3     public function init() {
4         require_once APPLICATION_PATH . '/configs/translations/pt_BR.php';
5         $translate = new Zend_Translate('array', $translationStrings, 'pt');
6         $this->setTranslator($translate);
7         $this->addElementPrefixPath('Biblioteca_Validate', 'Biblioteca/Validate/', 'validate');
8         $this->setName('emprestimo');
9         $id = new Zend_Form_Element_Hidden('id');
10        $data_emprestimo = new Zend_Form_Element_Text('data_emprestimo');
11        $data_emprestimo->setLabel('Data do Empréstimo:');
12        $data_emprestimo->setRequired(true);

```

```


13         ->addFilter('StripTags')
14         ->addFilter('StringTrim')
15         ->addValidator('NotEmpty')
16         ->addValidator('Date');
17     $membroFacade = new Biblioteca_Business_Facade_Membro();
18     $membros_options = $membroFacade->htmlselectBusiness();
19     $membro_id = new Zend_Form_Element_Select('membro_id');
20     $membro_id->addMultiOption('', 'Escolha um Membro');
21     if (sizeof($membros_options) > 0) {
22         foreach ($membros_options as $membro) {
23             $membro_id->addMultiOption($membro['id'], $membro['nome']);
24         }
25     }
26     $membro_id->setLabel('Membro:');
27         ->setRequired(true)
28         ->addFilter('StripTags')
29         ->addFilter('StringTrim')
30         ->addValidator('NotEmpty');
31     $valor_juros = new Zend_Form_Element_Text('valor_juros');
32     $valor_juros->setLabel('Valor do Juros: R$');
33         ->setRequired(true)
34         ->addFilter('StripTags')
35         ->addFilter('StringTrim')
36         ->addValidator('NotEmpty')
37         ->addValidator('Float');
38     $submit = new Zend_Form_Element_Submit('submit');
39     $submit->setLabel('Salvar');
40         ->setAttrib('id', 'submitbutton');
41     $this->addElements(array($id, $data_emprestimo, $membro_id, $valor_juros, $submit));
42 }
43 }

```

A parte de *interface* com o usuário foi estilizada com CSS, as figuras 7.5, 7.6 e 7.7 mostram telas referentes a listagem dos empréstimos, criação de um empréstimo e devolução de itens, respectivamente.

Membro	Data do Empréstimo	Valor do Juros	Excluir
Fernando Geraldo Mantoan	2009-11-07	R\$2	<a href="#">Excluir</a>

Figura 7.5: Tela de listagem de empréstimos.

 biblioteca

Logado como Administrador | Sair

Editoras

Autores

Livros

Membros

Empréstimos

Usuários

### Novo Empréstimo

< VOLTAR

Data do Empréstimo:

Membro:

Valor do Juros: R\$

Salvar

Figura 7.6: Tela de criação de um empréstimo.

 biblioteca

Logado como Administrador | Sair

Editoras

Autores

Livros

Membros

Empréstimos

Usuários

### Devolver Item

< VOLTAR

**Livro:** Zend Framework Componentes Poderosos para PHP

**Data do Empréstimo:** 2009-11-07

**Empréstimo realizado por:** Fernando Geraldo Mantoan

**Valor do Juros cobrado por cada dia de atraso:** R\$2

**Data Prevista para a Devolução:** 2009-11-16

Data de Devolução:

Valor Pago: R\$

Devolver Item

Figura 7.7: Tela de devolução de item de empréstimo.

## 8 *Considerações Finais*

O ciclo de vida de um *software* é composto de várias etapas, que vai desde sua composição até a manutenção do mesmo. Com a implementação utilizada neste trabalho pôde-se notar que as arquiteturas de *software* tornam a etapa de elaboração de um sistema muito mais organizada, pois partindo-se de convenções e de padronizações exigidas por ela, programadores saberão a forma de nomear componentes, identificar código e onde encontrar funcionalidades específicas.

Junto com os benefícios de uma arquitetura de *software*, a arquitetura proposta traz também diversos benefícios de reusabilidade, graças aos *design patterns*. Com eles, problemas que o sistema apresentou puderam ser resolvidos de uma maneira altamente aceita por programadores. Além destes *design patterns*, também foram utilizados *patterns* criados para melhorar a legibilidade e organização dos códigos, como o *Table Data Gateway* e *Data Mapper*, estes não pertencendo ao catálogo escrito pela GOF, que, em conjunto com o MVC separam a arquitetura em camadas lógicas.

Com isto, nota-se a extrema importância que uma arquitetura de *software* com *design patterns* tem no desenvolvimento de um sistema, porém ainda existe uma outra etapa do sistema que deve ser levada em consideração: a **manutenção**. Com a especificação dos *design patterns* e uma boa documentação da arquitetura, a manutenibilidade do sistema será muito mais viável, pois bastará ao desenvolvedor entender as regras da arquitetura e conhecer o conceito dos *patterns* utilizados para poder alterar ou incluir novos códigos a sistemas já prontos.

Pode-se concluir então que sistemas podem possuir um ciclo de vida bastante prolongado graças a união de arquiteturas de *software* e *design patterns*, beneficiando os clientes e, também os desenvolvedores.

## 8.1 Trabalhos Futuros

Devido a aplicação de *design patterns* depender muito do escopo do projeto, um dos trabalhos futuros é a seleção de novos *patterns*, ou a remoção dos definidos neste trabalho, para que a arquitetura solucione os problemas de projeto que possam surgir em outros *softwares*.

Além disso, desacoplar a arquitetura definida do Zend *Framework* também é um trabalho futuro. Assim, a mesma arquitetura poderia ser independente de *framework*, facilitando a migração entre os diversos *frameworks* disponíveis para PHP.

Outro trabalho futuro é modificar a estrutura da arquitetura para que ela seja baseada em *plugins*. Assim a tarefa de adicionar e editar funcionalidades se torna muito mais prática e consistente.



## *Referências Bibliográficas*

- APACHE. *Site Oficial do projeto Apache HTTP*. 2009. Disponível em: <<http://httpd.apache.org/>>. Acesso em: 21 nov. 2009.
- BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Boston: Addison Wesley, 2003.
- BUSCHMANN, F. et al. *Pattern-Oriented Software Architecture*. Chichester: Wiley, 1996.
- CAKESOFTWAREFOUNDATION. *Cake Software Foundation*. 2009. Disponível em: <<http://cakephp.org>>. Acesso em: 31 mai. 2009.
- CHANGEVISION. *Astah\* Site Oficial*. 2009. Disponível em: <<http://astah.change-vision.com>>. Acesso em: 21 nov. 2009.
- CODEIGNITER. *Code Igniter Site Oficial*. 2009. Disponível em: <[http://codeigniter.com/user\\_guide](http://codeigniter.com/user_guide)>. Acesso em: 21 nov. 2009.
- DALL'OGGIO, P. *PHP - Programando com Orientação a Objetos*. São Paulo: Novatec, 2007.
- FOWLER, M. et al. *Patterns of Enterprise Application Architecture*. Indianapolis: Addison-Wesley, 2002.
- GAMMA, E. et al. *Design Patterns, Elements of Reusable Object-Oriented Software*. Indianapolis: Addison-Wesley, 1995.
- GIL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. São Paulo: Atlas, 2002.
- HAYDER, H. *Object-Oriented Programming with PHP5*. UK: Packt Publishing, 2007.
- MELO, A. A. de; NASCIMENTO, M. G. F. do. *PHP Profissional*. São Paulo: Novatec, 2007.
- MYSQL. *MySQL Site Oficial*. 2009. Disponível em: <<http://www.mysql.com/about>>. Acesso em: 21 nov. 2009.
- PARRA, D. F.; SANTOS, J. A. *Metodologia Científica*. 4. ed. São Paulo: Futura, 2002.
- PHP. *Site Oficial da linguagem PHP*. 2009. Disponível em: <<http://www.php.net>>. Acesso em: 21 nov. 2009.
- PHPMYADMIN. *phpMyAdmin Site Oficial*. 2009. Disponível em: <<http://www.phpmyadmin.net>>. Acesso em: 21 nov. 2009.
- POPE, K. *Zend Framework 1.8 Web Application Development*. UK: Packt Publishing, 2009.

- PRESSMAN, R. S. *Engenharia de Software*. 5. ed. Rio de Janeiro: McGraw-Hill, 2002.
- SILVA, M. S. *Construindo sites com CSS e (X)HTML*. São Paulo: Novatec, 2008.
- SOMMERVILLE, I. *Engenharia de Software*. 6. ed. São Paulo: Addison Wesley, 2003.
- VAROTO, A. C. *Visões em Arquitetura de software*. São Paulo: [s.n.], 2002.
- XUE, Q.; ZHUO, X. W. *Prado QuickStart Tutorial*. 2009. Disponível em: <<http://www.pradosoft.com>>. Acesso em: 31 mai. 2009.
- ZEND. *Zend Framework*. 2009. Disponível em: <<http://framework.zend.com>>. Acesso em: 20 nov. 2009.