UNIVERSIDADE FEDERAL DE PELOTAS
Instituto de Física e Matemática
Departamento de Informática
Curso de Bacharelado em Ciência da Computação

Trabalho Acadêmico

Utilizando Padrões de Projeto no Desenvolvimento de Aplicações em Ambiente Web

Paulo Tiago Gomes Casanova

Pelotas, 2008

Paulo Tiago Gomes Casanova

UTILIZANDO PADRÕES DE PROJETO NO DESENVOLVIMENTO DE APLICAÇÕES EM AMBIENTE WEB

Trabalho acadêmico apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Prof[a]. Msc. Eliane da Silva Alcoforado Diniz.

Pelotas, 2008

Dados de catalogação na fonte:
Maria Beatriz Vaghetti Vieira – CRB-10/1032
Biblioteca de Ciência & Tecnologia - UFPel

C335u Casanova, Paulo Tiago Gomes

Utilizando padrões de projeto no desenvolvimento de aplicações em ambiente Web / Paulo Tiago Gomes Casanova ; orientador Eliane da Silva Alcoforado Diniz. – Pelotas, 2008. – 109f. - Monografia (Conclusão de curso). Curso de Bacharelado em Ciência da Computação. Departamento de Informática. Instituto de Física e Matemática. Universidade Federal de Pelotas. Pelotas, 2008.

1.Informática. 2.Engenharia de software. 3. Padrões de projeto. 4. Design Patterns. 5. Orientação a objetos. 6. PHP. I.Diniz, Eliane da Silva Alcoforado. II.Título.

CDD: 005.269

Agradecimentos

Quero agradecer, em primeiro lugar e acima de tudo, ao meu Deus ao qual sempre me deu forças e capacidade para seguir em frente diante das dificuldades. Também, gostaria de agradecer a minha amada noiva Lisa, pois sem sua ajuda, seu carinho, sua paciência, atenção e seu amor dedicados a mim, jamais teria conseguido realizar este trabalho, e não por menos, agradeço aos meus pais Adair e Raquel que sempre me deram todo o apoio e incentivo para estudar e seguir na minha profissão e também ao meus avós Edgar e Laci, pelo carinho e compreensão da minha longa jornada de estudos, que por vezes nos distanciava. Agradeço também aos meus irmãos Dudu, sua esposa Daniela e Bernardo, meu irmão mais novo, que pretende seguir a mesma faculdade, a todos eles que de uma forma ou de outra me ajudaram nesta caminhada, meu muito obrigado!

E, por último, mas não menos importante, agradeço a minha orientadora Eliane da Silva Alcoforado Diniz. Muito obrigado pela sua dedicação, amizade, apoio e competência. Espero ter atingido as tuas expectativas com este trabalho, professora.

“O temor do SENHOR é o princípio do conhecimento; os loucos desprezam a sabedoria e a instrução.”

Provérbios 1:7

Resumo

Padrões de projeto são entendidos como estruturas descritas em termos de objetos e classes que podem ser reutilizáveis para solucionar questões de implementação recorrentes em um projeto. Uma das maneiras de medir a qualidade de um sistema orientado a objetos é avaliar se os desenvolvedores tomaram cuidado com as colaborações comuns entre seus objetos. Focalizar-se em tais mecanismos durante o processo de desenvolvimento de um sistema, pode levar a uma arquitetura menor, mais simples e muito mais compreensível do que aquelas produzidas quando estes padrões são ignorados.

Neste cenário, este trabalho apresenta estudos de caso de aplicação de um grupo de padrões de projeto na solução de problemas de implementação recorrentes no desenvolvimento de aplicações em ambiente Web, utilizando ferramentas de programação orientadas a objetos. Como suporte para o trabalho, também, foi realizada uma pesquisa em algumas empresas de desenvolvimento Web na cidade de Pelotas, objetivando selecionar o referido grupo de padrões de projeto e analisar o atual estágio de aplicação destes padrões naquelas empresas.

Palavras-chave: Padrões de projeto. Design Patterns. Orientação a Objetos. Engenharia de Software. PHP.

Abstract

Design Patterns are understood like structures described in terms of objects and classes that can be reusable to resolve implementions questions that are applicants in a project. One way to measure the quality of an object-oriented system is evaluate if the developers take care of the common collaborations among their objects. The focus on these mechanisms during a system development process, can lead to a minor architecture more simple and more comprehensible than that produced when these patterns are ignored.

In this scenario, this work presents case studies that demonstrate applications of a group of design patterns in the solution of implementions problems common in the development of Web applications, using object-oriented programming tools. As support for the work, it has also performed a research in some web development companies in Pelotas, in order to select the Design Patterns group and analyze the current stage of application of these patterns in those companies.

Keywords: Desing Patterns. Object-oriented programming. Software Engineering. PHP

Lista de Abreviaturas

CI - Centro de Informática

Coinpel - Companhia de Informática de Pelotas

HTML - HyperText Markup Language

PHP - Hypertext Preprocessor

OOPSLA - Object-Oriented Programming, Systems, Languages & Applications

ADSWeb - Soluções de Tecnologia da Informação

TI - Tecnologia de informação

UFPEL - Universidade Federal de Pelotas

UML – Linguagem de Modelagem Unificada

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

A indústria de produção de software, principalmente, voltada para Web tem se desenvolvido bastante na última década e cada vez mais vem utilizando-se de recursos da Engenharia de Software. Atualmente, com a velocidade e quantidade que as informações surgem, os softwares necessitam ser mais robustos e ágeis para possibilitar uma boa manipulação e gerência dessas informações. Para satisfazer estas necessidades, é preciso desenvolver aplicações seguindo metodologias de Engenharia de Software que permitam um desenvolvimento otimizado, onde se diminua o tempo necessário de desenvolvimento, produza sistemas com baixa taxa de erros e com alta flexibilidade a alterações de requisitos.

Com a finalidade de atender esta demanda, intrínseca ao mercado de software, as empresas de desenvolvimento de aplicações direcionadas para Web têm dado maior enfoque a utilização de componentes reutilizáveis, frameworks e padrões que atendam aos objetivos de qualidade, rapidez e flexibilidade no desenvolvimento.

É neste cenário que padrões de projeto têm se tornado cada vez mais utilizados, provendo soluções para problemas recorrentes durante todas as fases do desenvolvimento, objetivando padronizar a metodologia de produção de software, aplicando as melhores práticas na solução de problemas de implementação, tornando as aplicações flexíveis a alterações de requisitos e servindo como base para o desenvolvimento de componentes e frameworks.

Uma maneira de se conseguir bons projetos arquiteturais é o reaproveitamento de experiências adquiridas. Segundo Gamma et al (2000, p. 18): “Uma coisa que os bons projetistas sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero. Em vez disso, eles utilizam soluções que já funcionaram no passado”. A questão central é de que forma reaproveitar estas experiências, sem utilizar-se de práticas relacionadas à transmissão informal dessas.

Para documentar estas soluções a ponto de reaproveitá-las, é necessário

primeiramente formalizá-las, transformando-as em padrões de projeto que possam ser compreendidos e reutilizados em outros projetos por outros desenvolvedores.

Neste trabalho, visando esta abordagem de utilização de padrões de projeto, são implementados estudos de caso de aplicação de alguns desses padrões no desenvolvimento de aplicações em ambiente Web. Para a seleção dos padrões abordados foi realizada uma pesquisa de campo em algumas empresas deste ramo na cidade de Pelotas. Ainda através desta pesquisa foi feita uma análise do atual estágio de utilização de padrões de projeto no desenvolvimento Web na referida cidade.

1.1 Motivação do trabalho

Neste trabalho vários fatores foram agentes motivadores, primeiramente o conhecimento e a experiência adquirida ao longo do desenvolvimento do trabalho. Principalmente, através da integração entre universidade e empresa que a realização da pesquisa de campo proporcionou, pois foram vários contatos e discussões com líderes de projeto de algumas empresas de desenvolvimento de software, contribuindo assim, para uma aproximação maior junto ao mercado de trabalho.

Outro fator motivador está relacionado aos resultados da pesquisa apresentada, que possibilitaram mostrar o atual estágio de utilização de padrões de projeto por parte de algumas empresas de desenvolvimento Web na cidade.

Também é destacado como fator motivacional os estudos de caso de aplicação prática dos padrões de projeto, o que possibilitou um aprimoramento do conhecimento técnico sobre o assunto. Todo profissional que atua ou deseja ingressar no mercado de tecnologia de informação - TI deve acompanhar as tendências e abordagens de desenvolvimento de software da área.

1.2 Objetivos do trabalho

Este trabalho tem como objetivo principal realizar estudos de caso de aplicação de um grupo de padrões de projeto, selecionados através de uma pesquisa de campo, na solução de problemas de implementação recorrentes no desenvolvimento de aplicações em ambiente Web.

Outro objetivo é a realização de uma análise do atual estágio de utilização destes padrões, por parte de algumas empresas de desenvolvimento Web na cidade

de Pelotas.

1.3 Apresentação do trabalho

Este trabalho está organizado da seguinte forma:

No capítulo 2, são abordados conceitos básicos como Orientação a Objetos, UML, frameworks e PHP. Estes conceitos são utilizados ao longo do trabalho e também, são de fundamental importância para o entendimento do mesmo.

No capítulo 3, é definido o que vem a ser padrões de projeto, apresentando suas principais características, benefícios e suas classificações.

No capítulo 4, é apresentado o catálogo completo de padrões de projeto, descrevendo as características básicas de cada padrão, incluindo diagramas UML que representam a estrutura usada no padrão.

No capítulo 5, são apresentados os resultados de uma pesquisa de campo, realizada com algumas empresas de desenvolvimento Web na cidade de Pelotas, objetivando analisar o atual estágio de utilização de padrões de projeto dessas empresas e também selecionar os padrões de projeto que são foco de estudo deste trabalho.

O capítulo 6 apresentará estudos de caso de aplicação de alguns padrões de projeto, selecionados com base na pesquisa de campo, no desenvolvimento de aplicações em ambiente Web.

No capítulo 7, é apresentado a conclusão do trabalho além de indicação para trabalhos futuros e alguns problemas encontrados ao longo do desenvolvimento do mesmo

2 Conceitos básicos

Neste capítulo são apresentados conceitos importantes para o estudo de padrões de projeto; estes conceitos serão utilizados ao longo do trabalho.

2.1 Orientação a objetos

2.1.1 Introdução

O paradigma da orientação a objetos, assim como outras abordagens, surgiu na tentativa de solucionar problemas existentes no desenvolvimento de software, principalmente com relação a custos e confiabilidade.

Todo software representa um modelo de um problema do mundo real (TURINE, 1998). O mundo real possui um espaço de problemas, formado por objetos que interagem. O modelo de software correspondente a esse problema é representado por um espaço de soluções. A orientação a objetos tenta diminuir o "*Gap* Semântico" existente entre o espaço de problemas e o espaço de soluções, por meio de um mapeamento dos objetos pertencentes ao espaço de problemas para o espaço de soluções, de tal maneira que operações sobre essas representações abstratas correspondam a operações do mundo real (Fig.2.1).

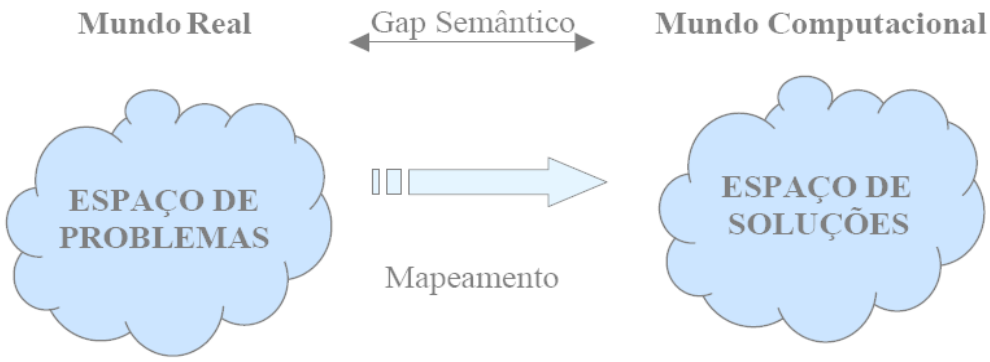

Figura 2.1 – Abordagem da orientação a objetos (TURINE,1998)

2.1.2 Classes e instâncias

A categorização dos objetos em grupos e/ou classes, baseada em propriedades comuns, é um dos princípios mais importantes do paradigma orientado a objetos; por exemplo, os objetos "Fiat Uno Placa: AAA-4444 Cor: Prata" e "Volkswagen Gol Placa: FFF-3333 Cor: Vinho" fazem parte da classe "Automóveis" e possuem propriedades comuns, tais como marca, modelo, placa, cor etc. Nesse caso, consideramos que "Fiat UnoPlaca: AAA-4444 Cor: Prata" e "Volkswagen Gol Placa: FFF-3333 Cor: Vinho" são instâncias, ou objetos, da classe "Automóvel" (Fig. 2.2).

Figura 2.2 – Classe e objetos

2.1.3 Atributos e Métodos

Os atributos são as propriedades da classe comuns a todos seus objetos ou instâncias. Em geral, os atributos são de uso exclusivo de cada classe, não podendo haver acesso a eles por objetos de outras classes; por exemplo, a classe “Automóvel” possui atributos Marca, Modelo, Placa e Cor.

Os métodos são procedimentos que atuam sobre um objeto ou sobre uma classe de objetos. Eles descrevem uma seqüência de ações a ser executada por um objeto, podendo fazer consultas ou alterações nos atributos desse objeto ou criar novos objetos. Através dos métodos especifica-se a um objeto como fazer alguma coisa. Os métodos são intrínsecos às classes e não podem ser separados; por exemplo, a classe “Automóvel” possui métodos Registrar, Eliminar, AlterarPlaca, etc. (Fig.2.3).

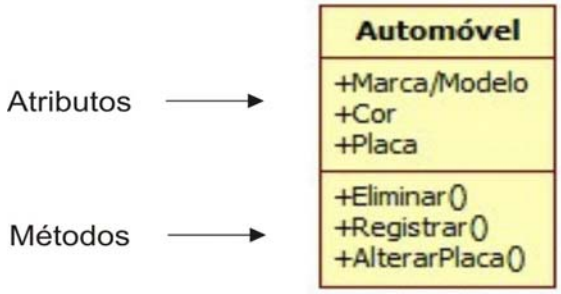

Figura 2.3 - Atributos e Métodos

2.1.4 Generalização, especialização e agregação

Os conceitos de generalização e especialização são decorrentes do conceito de classes, a partir de duas classes pode-se abstrair uma classe mais genérica. As subclasses satisfazem todas as propriedades das classes, essas subclasses constituem especializações. Deve existir pelo menos uma propriedade que diferencie duas classes especializadas; por exemplo, a partir de duas classes “Automóvel” e “Caminhão” pode-se criar a generalização “Veículo”, que possui todos os atributos comuns entre as especializações “Automóvel” e “Caminhão” (Fig.2.4). Nesse caso, diz-se que Veículo é a superclasse (generalização) e que “Automóvel” e “Caminhão” são subclasses (especializações).

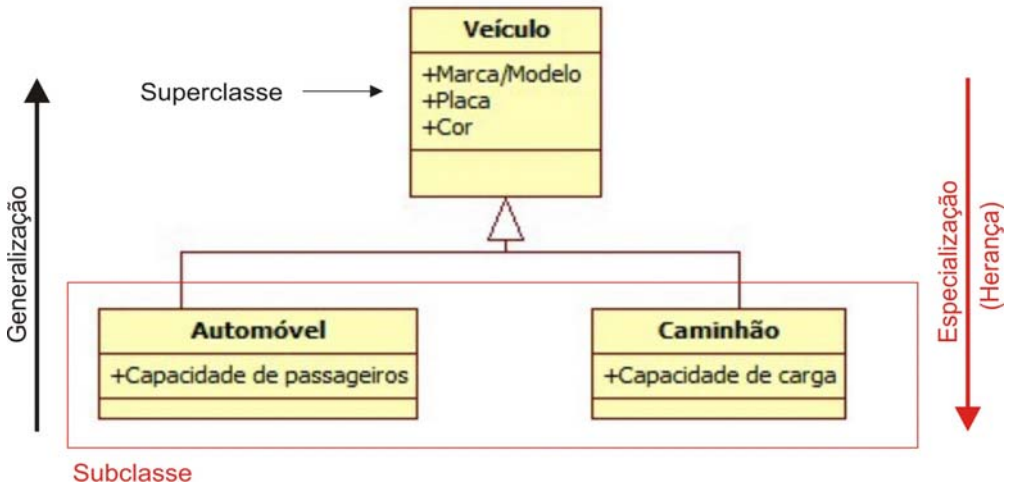

Figura 2.4 – Generalização/Especialização

A agregação ou decomposição consiste na composição de uma nova classe como um agregado de classes pré-existentes. É freqüentemente referida por relacionamento “todo-parte”, no qual o agregado (“todo”) é composto de partes

(COOK, 1994); por exemplo, na Fig.2.5 a classe “Livro” é agregada à classe “Capítulo”, o que significa que cada capítulo é parte de um único livro. O livro é formado por seus capítulos e se um livro for destruído, assim o serão seus capítulos.

Figura 2.5 - Agregação

2.1.5 Classes Abstratas X Classes Concretas

As classes abstratas definem seus métodos em termos de outros métodos indefinidos que precisam ser implementados por suas subclasses concretas; por exemplo, a classe “Coleção” (Fig. 2.6) define métodos como Selecione, Recupere, Inclua, etc. em termos de um método de iteração Faça. Suas subclasses “Matriz”, “Conjunto” e “Dicionário” definem o método Faça e assim podem usar os métodos herdados de “Coleção”. Nesse caso, o método Faça é reescrito em cada uma das subclasses concretas, enquanto os métodos Selecione, Recupere e Inclua são métodos definidos apenas na classe abstrata e herdados pelas subclasses. Nada impede que uma das classes especialize um dos métodos da classe. A classe “Coleção” nunca possui instâncias, as instâncias são sempre de umas das subclasses concretas “Matriz”, “Conjunto” ou “Dicionário”.

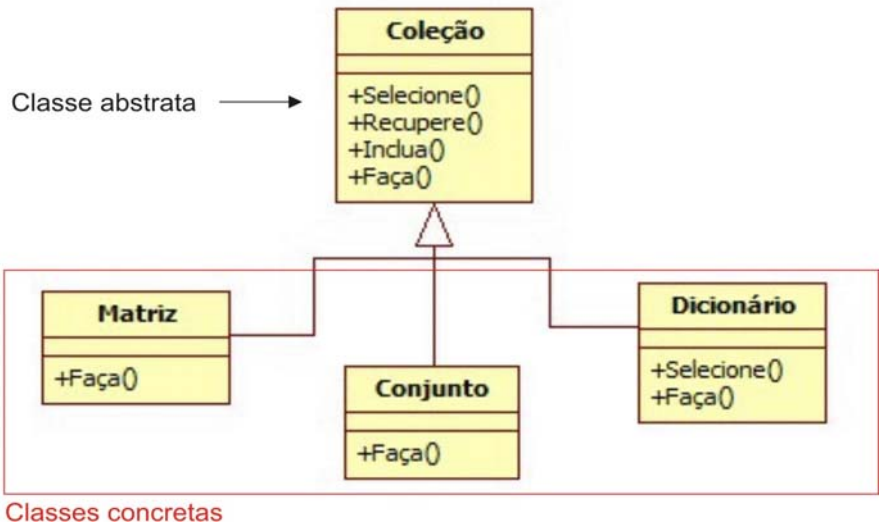

Figura 2.6 - Classes Abstratas e concretas

2.1.6 Herança

A herança é um mecanismo que permite definir uma nova classe (subclasse) a partir de uma classe já existente (superclasse). Ao se estabelecer uma Especialização (subclasse) de uma classe, a subclasse herda as características comuns da superclasse, isto é, os atributos e métodos da superclasse passam a fazer parte da especificação dos atributos e dos métodos da subclasse.

> Herança é uma forma de reutilização de software em que novas classes são criadas a partir de classes existentes, absorvendo seus atributos e comportamentos e sofisticando-os com capacidades que as novas classes exigem (DEITEL, 2001).

A subclasse pode adicionar novos métodos, como também reescrever métodos herdados. Na Fig.2.6, as subclasses Matriz, Conjunto e Dicionário herdam os métodos Selecione, Recupere e Inclua da superclasse Coleção e reescrevem o método Faça. A classe Dicionário reescreve também o método Selecione.

2.1.7 Encapsulamento e mensagens

> Encapsulamento é a maneira de se associar os dados e as operações executadas neles de forma organizada, especificando claramente quais delas podem ser executadas em quais objetos (LINDEN, 1997).

Na orientação a objetos, as estruturas de dados ou atributos das classes ficam encapsuladas, ou seja, só podem ser manipuladas pelos próprios métodos da classe. O objeto possui uma parte privada (Visão Interna) e uma parte compartilhada (Visão Externa ou Interface). O encapsulamento restringe a visibilidade do objeto mas facilita o reuso, pois os atributos e os métodos são empacotados sob um nome e podem ser reusados como uma especificação ou componente de programa.

A passagem de mensagens é o mecanismo através do qual os objetos se comunicam, invocando as operações desejadas. As operações ou métodos que podem ser invocados por outras classes fazem parte da interface ou visão externa da classe. A interface especifica quais operações sobre os objetos da classe podem ser realizadas, mas não como a operação será executada.

2.1.8 Polimorfismo

Ao receber uma mensagem para efetuar uma operação, é o objeto quem determina como a operação deve ser efetuada, pois ele tem comportamento próprio. Como a responsabilidade é do receptor e não do emissor, pode acontecer que uma mesma mensagem ative métodos diferentes, dependendo da classe de objetos para onde é enviada a mensagem.

Por meio do polimorfismo, é possível criar várias classes com interfaces idênticas, porém com objetos e implementações diferentes. Assim, uma mensagem pode ser executada de acordo com as características do objeto que está recebendo o pedido de execução do serviço; por exemplo, o operador “+” é um exemplo clássico do polimorfismo mesmo nas linguagens mais antigas: ele é usado tanto com parâmetros numéricos quanto com parâmetros do tipo caractere ou outros. Outro exemplo de polimorfismo é uma lista genérica que pode conter elementos de qualquer tipo como inteiros, *strings, arrays*, etc.

2.2 Métodos de análise e projeto orientado a objetos

Além da programação orientada a objetos, difundida no início da década de 80 pelo surgimento de linguagens como Smalltalk e Ada, diversos métodos de análise e projeto orientado a objetos têm sido propostos nos últimos anos, entre eles:

- CRC (*Class Responsability Collaborator*, Beecke e Cunningham, 1989)
- OOA (*Object Oriented Analysis*, Coad e Yourdon, 1990)
- *Booch* (Booch, 1991)
- OMT (*Object Modeling Technique*, Rumbaugh, 1991)
- *Objectory* (Jacobson, 1992)
- *Fusion* (Coleman etal, 1994)

O objetivo desses métodos é tornar a análise compatível com a codificação do software, o que não seria efetivo se métodos baseados em outros paradigmas fossem aplicados em virtude da grande quantidade de notações diferentes para representação de software orientado a objetos, foi proposta uma notação universal, chamada *Unified Modeling Language* - UML (ERIKSSON, 1997). Essa notação, será utilizada no decorrer deste trabalho para representar os diversos diagramas relativos aos padrões de projeto.

2.3 *Framework*

Framework é descrito como um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas (COAD, 1992) e também é considerado como um conjunto de classes abstratas e concretas que provêem uma estrutura genérica de soluções para um conjunto de problemas (JOHNSON,1998). Essas classes podem fazer parte de uma biblioteca de classes ou podem ser específicas da aplicação. *Frameworks* possibilitam reutilizar não só componentes isolados, como também toda a arquitetura de um domínio específico e podem ser compostos por um conjunto de padrões de projeto. Na pesquisa de campo realizada neste trabalho é importante ter este conceito definido.

2.4 PHP

O *Hypertext Preprocessor* - PHP é uma linguagem voltada para o desenvolvimento Web, que foi desenvolvida para trabalhar em conformidade com a estrutura cliente-servidor. Nessa estrutura, o servidor é o responsável por interpretar os *scripts* que compõem o documento solicitado, transformá-los em código *HyperText Markup Language* - HTML e enviar o resultado ao cliente, que fez a solicitação. Dessa forma, O PHP permite a criação de sistemas Web dinâmicos com a utilização de consultas a banco de dados, controle de *cookies* e variáveis de seção, dentre muitos outros recursos. A Fig.2.7 ilustra o funcionamento desta tecnologia.

Seguindo a linha adotada pela maior parte das linguagens de programação, o PHP aos poucos foi agregando os conceitos da Orientação a Objetos. Esta adequação culminou no desenvolvimento da versão 5, que engloba a maior parte destes conceitos. Essa adequação possibilitou aos desenvolvedores usufruir das principais vantagens advindas com a orientação a objetos no processo de desenvolvimento de aplicações, tais como reutilização de código, métodos construtores e destrutores, herança e encapsulamento.

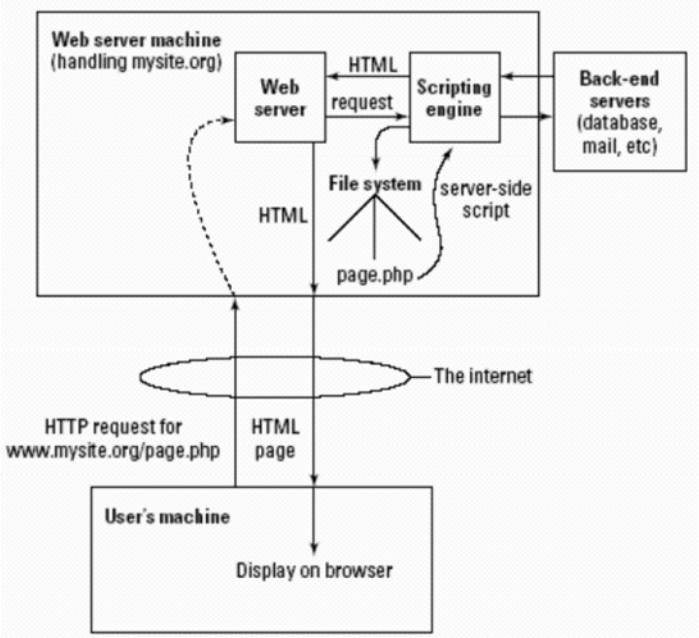

Figura 2.7 - Tecnologia cliente-servidor (MORGAN;PARK;CONVERSE, 2004)

Dessa forma, a camada de negócios de uma aplicação Web passa a ser trabalhada tal como seria desenvolvida em um aplicativo desktop, se apoiando em padrões de desenvolvimento que facilitem a manutenção e a reusabilidade de código. Esta linguagem de programação será utilizada neste trabalho para mostrar como os padrões de projeto, abordados no capítulo 6, são aplicados na prática no desenvolvimento de aplicações Web.

3 Padrões de projeto

3.1 Introdução

O arquiteto Christopher Alexander em seus livros: *A Pattern Language*[1] e *The Timeless Way of Building*[2] utiliza o termo *pattern* para referenciar estruturas que podem ser utilizadas para solucionar problemas, que ocorrem seguidamente na área onde o escritor atua. Apesar destes livros tratarem sobre arquitetura e planejamento urbano as idéias que eles trazem sobre estes assuntos podem ser aplicadas em muitas outras áreas, inclusive no desenvolvimento de software.

A aplicação destas idéias na área de desenvolvimento de software, surgiu no ano de 1987 quando Kent Beck e Ward Cunninghan estavam trabalhando com Smalltalk em projetos de interface gráfica e enfrentavam problemas para terminar um projeto. Então, resolveram catalogar cinco padrões de projeto para evitar que jovens programadores cometessem erros já conhecidos por programadores mais experientes. Além de possibilitar que o projeto no qual eles trabalhavam tivesse um ritmo de desenvolvimento mais rápido. A partir daí, a idéia de utilizar padrões para desenvolver software começou a crescer, com cada vez mais pessoas se interessando e estudando sobre o assunto.

No ano de 1994, no evento *Object-Oriented Programming, Systems, Languages & Applications* - OOPSLA, foi apresentado o livro intitulado "*Design Patterns: Elements of Reusable Object-Oriented Software"*. Os autores[3] deste livro se intitularam "*Gang Of Four"* sendo muitas vezes conhecidos por este nome. Esta referida obra, constitui-se o eixo principal de base teórica referencial para todo este trabalho. Contudo também são feitos uso de outras fontes que dão maior suporte e legitimidade para toda esta pesquisa.

---

[1] A Pattern Language Towns, Buildings, Construction (Oxford University Press, 1977).
[2] The Timeless Way of Building (Oxford University Press, 1979).
[3] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides.

3.2 O que é um padrão de projeto?

A grande maioria dos desenvolvedores de software, já experimentaram a sensação de estar frente a um determinado problema, do qual já tinham enfrentado anteriormente e que de alguma forma o resolveram, porém não lembram mais como o fizeram. É exatamente neste momento que entra padrões de projeto, pois é inaceitável, que toda vez, ao se deparar com uma determinada situação, que seguidamente se repete e que já se tenha resolvido antes, tenha-se que pensar novamente na busca de uma solução para ela. Padrões de projeto se propõem a combater este sentimento de que: Quem desenvolve software tem muitas vezes a sensação de estar reinventando a roda (SUN MYCROSYSTEM, 2002).

Profissionais experientes sempre reutilizam suas experiências passadas para resolver problemas já conhecidos, de forma que não tenham que redescobrir a solução a cada vez (GAMMA, 2000; ALUR, 2002; MESTKER, 2002). Projetistas familiarizados com certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los (GAMMA, 1995). Padrões de projeto nada mais são, então, do que uma descrição de como resolver, da melhor forma possível, um determinado problema que se repete.

A seguir, é apresentada a definição de padrões de projeto dada por alguns dos mais importantes autores nesta área:

De acordo Gamma (2000), uma definição para padrões de projeto seria: “Descrição de objetos de comunicação e classes que são customizados para resolver um problema geral de design em um contexto particular”.

Alexander (1977) define um padrão como “uma construção de três partes. Primeiro vem o contexto, isto é, sobre quais condições este padrão é empregado. Depois são as “forças do sistema”. A terceira parte é a solução: a configuração que equilibra o sistema de forças ou resolve o problema apresentado”.

Riehle e Zullighoven (1996) definem um padrão como sendo: “uma abstração de uma forma concreta que se repete em um contexto específico não-arbitrário”.

De acordo com Richard (1996) cada padrão é uma regra de três partes, uma que expressa uma relação entre um certo contexto, um certo sistema de forças que ocorre repetidamente nesse contexto, e uma certa configuração de software que permite que estas forças sejam resolvidas.

Coplien (1996) afirma que: “padrões é uma recente engenharia de software

para resolver problemas que nasceram a partir da comunidade de orientação a objeto".

Fowler (1996) define: "padrão é uma idéia que foi útil em um contexto e, provavelmente, será útil em outros".

Embora cada autor tenha uma forma particular de expressar sua idéia sobre padrões de projeto, todas elas vêm ao encontro da definição formulada por Gamma (2000).

3.3 Descrevendo os padrões de projeto

Para descrever os padrões de projeto Appleton (1997) e Gamma (2000), propõem um formato consistente onde cada padrão é dividido em seções de acordo com o modelo que será apresentado na Tab. 3.1. Este modelo fornece uma estrutura uniforme às informações, tornando os padrões de projeto mais fáceis de aprender, comparar e de se utilizar.

Tabela 3.1 – Modelo para descrever padrões de projeto

Nome e classificação do padrão	O nome do padrão expressa a sua própria essência de forma sucinta. Um bom nome é vital, porque ele se tornará parte do seu vocabulário de projeto.
Intenção e objetivo	É uma curta declaração que responde às seguintes questões: o que faz o padrão de projeto? Quais os seus princípios e sua intenção? Que tópico ou problema particular ele trata?
Também conhecido como	Outros nomes bem conhecidos para o padrão, se existirem.
Motivação	Um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão solucionam o problema. O cenário ajudará a compreender as descrições mais abstratas do padrão que vêm a seguir.
Aplicabilidade	Quais são as situações nas quais o padrão de projeto pode ser aplicado? Que exemplos de maus projetos ele pode tratar? Como você pode reconhecer essas situações?
Estrutura	Uma representação gráfica das classes do padrão usando uma notação de modelagem, UML por exemplo.
Participantes	As classes e/ou objetos que participam do padrão de projeto e suas responsabilidades.

Colaborações	Como as classes participantes colaboram para executar suas responsabilidades.
Conseqüências	Como o padrão suporta a realização de seus objetivos? Quais são os seus custos e benefícios e os resultados da sua utilização? Que aspecto de um sistema ele permite variar independentemente?
Implementação	Sugestões ou técnicas que precisa-se conhecer sobre a implementação do padrão? Existem considerações específicas de linguagem?
Exemplo de código	Fragmentos de código que ilustrem como você pode implementar o padrão.
Usos conhecidos	Exemplos do padrão encontrados em sistemas reais.
Padrões relacionados	Que padrões de projeto estão intimamente relacionados com este? Quais são as diferenças importantes? Com quais outros padrões este deveria ser usado?

FONTE: baseado em (GAMMA , 2000)

3.4 Classificação dos padrões de projeto

Segundo Gamma (2000), os padrões de projetos são classificados por dois critérios. Sendo o primeiro chamado de finalidade e o segundo de escopo (Tab. 3.2).

O critério de finalidade indica o que um padrão faz. Os padrões podem ter finalidade de criação, estrutural ou comportamental. Padrões de criação preocupam-se com o processo de criação de objetos. Os padrões estruturais lidam com a composição de classes ou de objetos. Já os padrões comportamentais caracterizam a forma com que as classes ou objetos interagem e distribuem responsabilidades.

O outro critério chamado de escopo, específica se o padrão é aplicado a classes ou a objetos. Os padrões para classe lidam com os relacionamentos entre classes e suas subclasses. Os padrões para objetos lidam com relacionamentos entre objetos.

Tabela 3.2 – Lista dos padrões de projeto

		Finalidade		
		Criação	Estrutural	Comportamental
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweygth Proxy	Chain of Responsibility Command Iterator Mediator Observer State Strategy Visitor

FONTE: baseado em (GAMMA, 2000)

3.5 Motivos para se adotar padrões de projeto

A utilização de padrões de projeto no desenvolvimento de sistemas em geral, proporciona vários benefícios, onde Alur, Crupi e Malks (2001) e Arrington (2002), destacam:

- Uma forma de medir a qualidade de um sistema orientado a objetos é avaliar se os desenvolvedores tomaram bastante cuidado com as colaborações comuns entre seus objetos. Focalizar-se em tais mecanismos durante o desenvolvimento de um sistema pode levar a uma arquitetura menor, mais simples e muito mais compreensível do que aquelas produzidas quando estes padrões são ignorados;
- Alavancar o uso de uma solução comprovada. Após comprovado o sucesso de um padrão, este pode ser reutilizado em tantos outros projetos futuros quantos forem necessários, prevenindo redundâncias comuns em projetos de software;
- Prover um vocabulário comum. Geralmente um projeto de software é desenvolvido por uma equipe e a comunicação é crucial para o sucesso deste projeto. Padrões de projeto podem otimizar a comunicação entre a equipe, melhorar a troca de idéias e aumentar o nível de abstração, assim diminuindo o risco de interpretações errôneas;
- Restringir o domínio da solução. Padrões de projeto evitam o emprego de soluções muito sobrecarregadas, levando o projeto a um caminho de

soluções mais simples e rápidas;

- Facilidade de uso. Desde o início do movimento existe um grande esforço para promover a documentação dos padrões criados, englobando as mais diversas áreas de aplicação, assim como um esforço para tornar essa documentação acessível;
- Melhora a documentação de sistemas. Uma das características marcantes nos Padrões de projeto é que eles são bem documentados. Um padrão é composto por várias partes, todas estas bem documentadas para mostrar como ele faz aquilo que se propõe, sob qual contexto ele é utilizado e quais as conseqüências da utilização do mesmo.
- Facilidade de manutenção. A manutenção é um processo que consome uma grande quantidade de tempo e recursos no ciclo de vida do software. Uma das maneiras de se minimizar esta etapa é fazendo bem as etapas anteriores. A utilização de Padrões de projeto no processo de produção de software contribui para minimizar o esforço com manutenção no futuro, pois um dos pontos fortes dos padrões de software e a documentação e também suas soluções propostas são comprovadamente de qualidade.

3.6 Como utilizar um padrão de projeto

Depois de identificado o contexto do problema e verificado a existência de um padrão que possa ser utilizado para ajudar a resolvê-lo, deve-se selecionar o padrão. Uma vez escolhido o padrão de projeto, pode-se seguir uma abordagem passo a passo para efetivamente aplicar o padrão (GAMMA et al, 2000), esta abordagem e descrita a seguir:

1. Ler o padrão por inteiro uma vez, assim obter uma visão geral e assegurar-se que o padrão de projeto selecionado é correto para o problema em questão;
2. Certifique-se de que compreende as classes e objetos do padrão e como se relacionam entre si;
3. Olhe um exemplo de código do padrão, isso ajuda a aprender como implementar o padrão;

4. Escolher nomes para os participantes do padrão que tenham sentido no contexto da aplicação, assim tornando o padrão mais explicito na implementação;
5. Definir e Identificar as classes existentes na aplicação que serão afetadas pelo padrão;
6. Definir nomes específicos da aplicação para as operações no padrão. Os nomes em geral dependem da aplicação. Ser consistente, nas convenções de nomenclatura;
7. Implementar as operações para suportar as responsabilidades de colaborações presentes no padrão. Consulte a seção de implementação para verificar os exemplos de código do padrão.

4 Catálogo de padrões de projeto

Um catálogo de padrões é uma coletânea de padrões relacionados. Em geral subdivide os padrões em um pequeno número de categorias abrangentes e pode incluir algumas referências cruzadas entre os padrões (APPLETON, 1997). Essa estrutura oferece um esquema de classificação e recuperação de seus padrões, já que eles estão subdivididos em categorias e adiciona uma maior estrutura e organização a essa coleção de padrões. Neste capitulo, é apresentada uma visão geral do catalogo de padrões de projeto proposto por (GAMMA, 2000) e abordado por Metsker (2004).

4.1 Padrões de criação

Os padrões de criação abstraem o processo de instanciação. Ajudando a tornar um sistema independente da forma como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto. Segundo Metsker (2004) “Os Padrões orientados para criação permitem a construção de um novo objeto mediante outros meios que não a chamada de um método construtor de classes”.

4.1.1 Abstract Factory

O *Abstract Factory* é um padrão de criação que tem como objetivo criar uma família de objetos relacionados ou dependentes sem a necessidade de especificar a classe concreta destes objetos. Em outras palavras, este padrão é uma fábrica de objetos que retorna uma das várias fábricas.

Muitas vezes um programa necessita criar famílias inteiras de objetos que se relacionam apenas entre si. As diferentes famílias de objetos apresentam mesma estrutura comum, mas não se relaciona com as outras famílias.

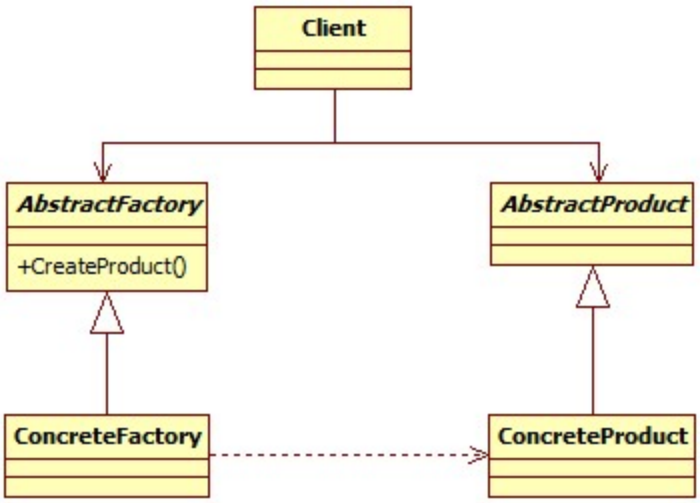

Figura 4.1 – Estrutura do *Abstract Factory*

Na Fig.4.1 a classe *AbstractFactory* é responsável pela declaração de uma interface para operações (métodos) de criação de objetos-produtos abstratos. O *ConcreteFactory* faz a implementação das operações que criam objetos-produtos concretos da classe *Product*.

A classe *AbstractProduct* faz a declaração de uma interface para um tipo de objeto-produto, onde nesta interface é definido uma classe de produto concreto para ser criado pela *ConcreteFactory* correspondente. Implementando a interface da classe *AbstractProduct*. Enfim a classe *Client* utiliza-se das interfaces criadas através das classes *AbstractFactory* e *AbstractProduct*.

Tabela 4.2 - Quando utilizar o *Abtract Factory*

Aplicabilidade	• Um sistema deve ser independente de como seus produtos são criados, compostos e representados; • Um sistema deve ser configurado com uma entre várias famílias de produtos; • Uma família de objetos-produto for projetada para ser usada em conjunto e você necessita garantir esta restrição; • Quer-se fornecer uma biblioteca de classes de produtos e quer revelar somente suas interfaces, não suas implementações.

4.1.2 Factory Method

O *Factory Method* é um padrão de projeto que tem como finalidade criar uma interface para instanciação de objetos que mantém isoladas as classes concretas usadas da requisição da criação destes objetos. Definindo uma interface para a criação de objetos, porém deixa as subclasses decidirem qual classe irão instanciar. O *Factory Method* permite que uma classe transfira para as subclasses a responsabilidade pela criação de novas instâncias.

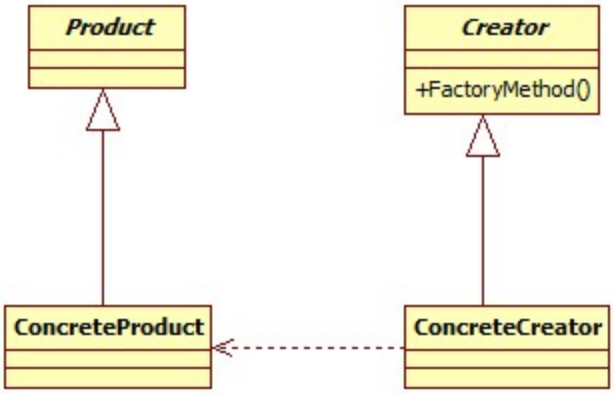

Figura 4.3 – Estrutura do *Factory Method*

Na Fig.4.3 a classe *Product* define a interface dos objetos que o método *FactoryMethod* cria. O *ConcreteProduct* implementa a interface de *Product*. A classe *Creator* declara o *FactoryMethod*, que retorna um objeto do tipo *Product*. O *Creator* pode também definir uma implementação padrão do *FactoryMethod* que irá retornar um *ConcreteProduct* específico. Além de poder chamar o *FactoryMethod* para criar um objeto do tipo *Product*. O *ConcreteCreator* sobrepõem o *FactoryMethod* a fim de retornar uma instância de um produto concreto.

Tabela 4.4 - Quando utilizar o *Factory Method*

Aplicabilidade	• Uma classe não pode antecipar qual a classe do objeto que ela deve criar; • Uma classe quer que suas subclasses definam o objeto que elas criam; • Classes delegam responsabilidades para uma entre várias subclasses auxiliares, e pretendesse limitar o conhecimento de qual subclasse auxiliar recebeu a delegação.

4.1.3 Builder

O *Builder* é padrão que tem como objetivo separar a construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações.

Um programa muitas vezes pode necessitar de fazer a leitura de um formato de documento (fonte) e convertê-lo em vários outros formatos diferentes (objeto). Se os formatos (objeto) não são definidos anteriormente, é possível configurar o programa com um conversor (builder) que pode ser especializado em diferentes formatos e operações de conversão.

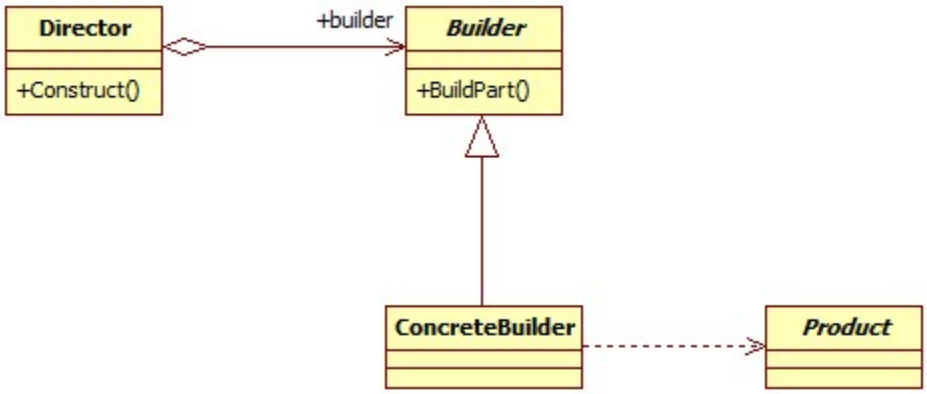

Figura 4.5 – Estrutura do *Builder*

Na Fig.4.5 a classe *Builder* declara uma interface abstrata que é responsável pela criação das partes do objeto da classe *Product*. O *ConcreteBuilder* é responsável por construir e montar as partes do *Product* através da implementação da interface *Builder*. Além de definir e manter o controle da implementação que ele construiu, fornecendo uma interface para retornar o *Product*. A classe *Director* é responsável pela construção de um objeto utilizando a interface *Builder*. Finalmente, a classe *Product* representa o objeto complexo que está sendo construído.

Tabela 4.6 - Quando utilizar o *Builder*

Aplicabilidade	• O algoritmo para a criação de um objeto complexo deve ser independente das partes que o compõem e de como estas são conectadas entre si; • O processo de construção deve permitir a criação de diferentes representações do objeto construído.

4.1.4 Prototype

O *Prototype* é um padrão de projeto que tem como finalidade fornecer uma outra forma de se construir objetos de tipos arbitrários. Especificando o tipo de objeto a ser criado através de uma instância-protótipo e criar novos objetos através da cópia deste protótipo. Assim, evitando a criação de várias instâncias da mesma classe devido ao custo da criação de uma instância e também evitar a criação de fábricas semelhantes onde à diferença está apenas nos tipos diferentes de objetos que a fábrica cria.

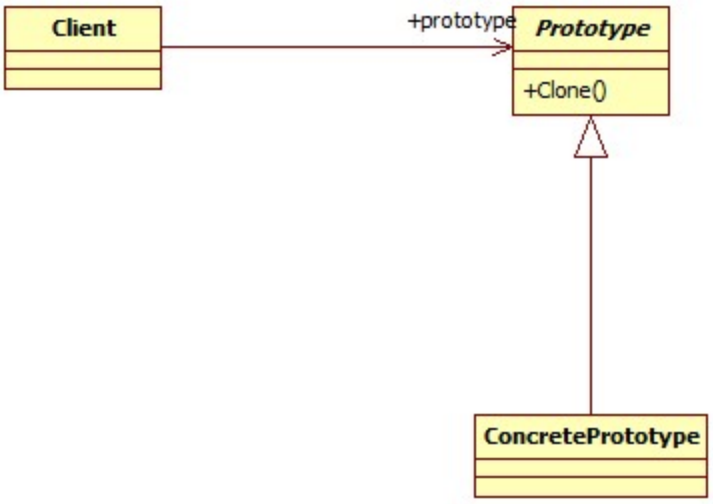

Figura 4.7 – Estrutura do *Prototype*

Na Fig.4.7 a classe *Prototype* define uma interface para permitir a própria clonagem. O *ConcretePrototype* implementa a operação de clonagem e o *Client* cria um novo objeto, solicitando o clone ao *Prototype*.

Tabela 4.8 - Quando utilizar o *Prototype*

Aplicabilidade	• O sistema deve ser independente de como os seus produtos são criados, compostos e representados; • As classes a instanciar são especificadas em tempo de execução; • E pretendido evitar a construção de uma hierarquia de classes de fabricação paralela a uma hierarquia de classes de produtos por elas fabricados.

4.1.5 Singleton

O padrão de projeto *Singleton* tem como objetivo garantir que para uma classe específica, deva existir somente uma única instância dela, a qual deva ser acessível de forma global e uniforme no sistema. Garantindo a existência de apenas um objeto, independentemente do número de requisições que receber para criá-la.

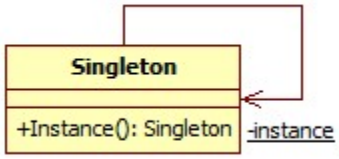

Figura 4.9 – Estrutura do *Singleton*

Na Fig.4.9 a classe *Singleton* define um método chamado *instance,* permitindo aos clientes acessar sua única instância. Responsável por criar e manter sua única instância.

Tabela 4.10 - Quando utilizar o *Singleton*

Aplicabilidade	• Deve haver exatamente uma única instância de uma classe e ela deve estar disponível a todos os clientes de um ponto de acesso bem definido; • Deseja-se que a única instância possa ser estendida por herança e os clientes serem capazes de utilizar essa instância estendida sem terem de modificar o seu código.

4.2 Padrões estruturais

Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Nos padrões estruturais de classes é utilizada a herança para compor interfaces ou implementações, o resultado é uma classe que combina as propriedades das suas classes ancestrais. Nos padrões estruturais de objetos é descrito maneiras de compor objetos para obter novas funcionalidades. Segundo Lozano (2003) “Os Padrões estruturais definem o relacionamento entre elementos de informação ou componentes de uma aplicação”.

4.2.1 Adapter

O padrão de projeto *Adapter* tem como finalidade permitir que dois objetos se comuniquem mesmo que tenham interfaces incompatíveis. Algumas vezes uma classe não é reusável, por existir uma incompatibilidade com a interface de uma aplicação de um domínio específico. A solução é a criação de um objeto adaptador, no qual encapsule e filtre as especificidades da classe adaptada.

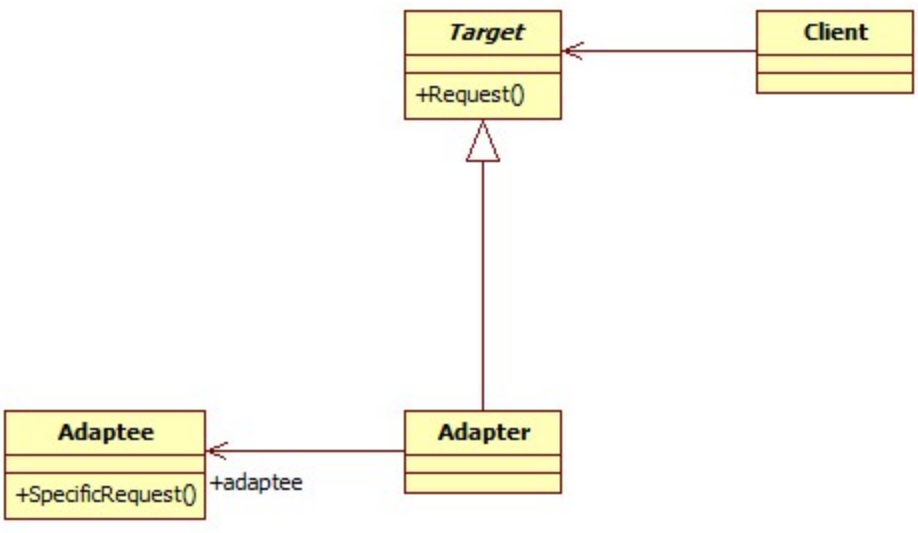

Figura 4.11 – Estrutura do *Adapter*

Na Fig.4.11 o padrão de projeto Adapter utiliza a classe *Target,* que define a interface de domínio específico que o cliente utiliza. Na classe *Adapter* é realizado a adaptação da interface *Adaptee* para ser utilizada pela classe *Target*. O *Adaptee* define uma interface pré-existente que necessita ser adaptada. Finalmente a classe *Client* colabora entre os objetos conforme a interface *Target*.

Tabela 4.12 - Quando utilizar o *Adapter*

Aplicabilidade	• Quando se quer usar uma classe existente, e sua interface não é compatível com uma que foi criada; • Quando se quer criar uma classe reusável que vai cooperar com classes não relacionadas ou não previstas antes, isto é, classes que não apresentam necessariamente a mesma interface.

4.2.2 Composite

O padrão *Composite* tem como objetivo lidar com uma estrutura de elementos agrupada hierarquicamente, permitindo que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.

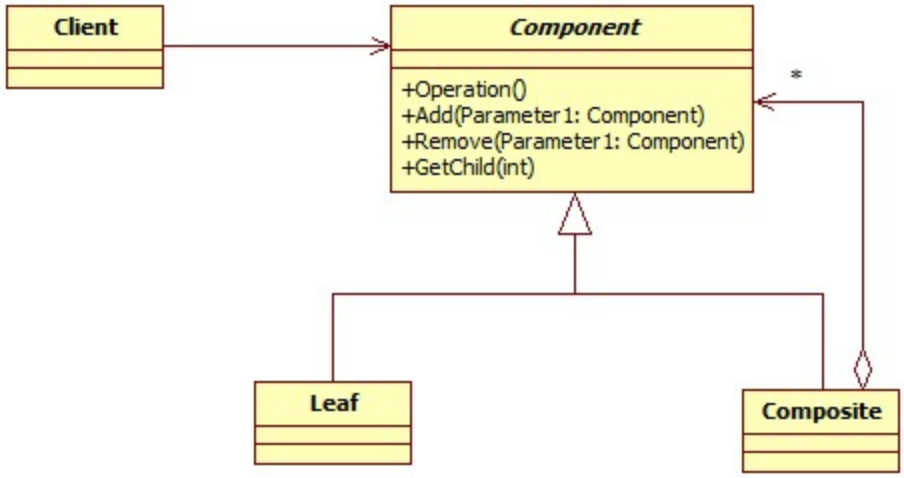

Figura 4.13 – Estrutura do *Composite*

Na Fig.4.13 a classe *Component* é responsável por declarar a interface para objetos da composição. Esta classe faz a implementação do comportamento padrão que seja comum a todas as classes. Declarando uma interface para acessar e gerenciar os seus elementos filhos. Também define uma interface para acessar o pai de um componente na estrutura recursiva, implementando-a se for necessário. A classe *Leaf* representa objetos folha na composição, que são aqueles que não possuem filhos, definindo um comportamento para objetos primitivos da composição. Na classe *Composite* é definido o comportamento para componentes que possuem filhos. Esta classe implementa as operações relacionadas aos filhos que foram definidas na interface de *Component*. O *Client* manipula objetos da composição

através da interface *Component*.

Tabela 4.14 - Quando utilizar o *Composite*

Aplicabilidade	• Quer representar hierarquias de objeto parte-todo; • Quer clientes aptos a ignorar as diferenças entre composições de objetos e objetos individuais. Clientes tratarão objetos de modo uniforme.

4.2.3 Decorator

O padrão de projeto *Decorator* tem como finalidade atribuir responsabilidades adicionais a um objeto dinamicamente fornecendo uma alternativa flexível a subclasses para a extensão da funcionalidade.

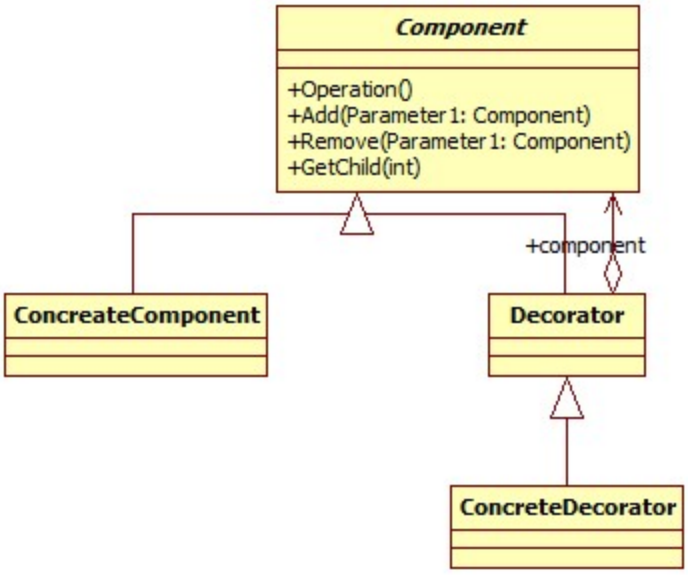

Figura 4.15 – Estrutura do *Decorator*

Na Fig.4.15 a classe *Component* define a interface de objetos que podem ter responsabilidades adicionadas dinamicamente. *Component* é uma classe abstrata para fatorar o código comum do *ConcreteComponent*. A classe *ConcreteComponent* define um objeto ao qual se podem adicionar responsabilidades adicionais. O *Decorador* mantém uma referência a um objeto *Component* e define uma interface igual à do *Component*. A classe *ConcreteDecorador* adiciona responsabilidades ao *Component*.

Tabela 4.16 - Quando utilizar o *Decorator*

Aplicabilidade	• Para adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, sem afetar outros objetos; • Quando extensão através de herança é impraticável. Algumas vezes uma grande quantidade de extensões independentes são possíveis e seria necessário um imenso número de subclasses para suportar cada combinação possível entre elas; • Quando uma definição de classe pode estar escondida ou não disponível para herdar.

4.2.4 Bridge

O padrão de projeto *Bridge* tem como finalidade separar uma abstração de sua própria implementação de forma que as duas possam mudar de forma independente.

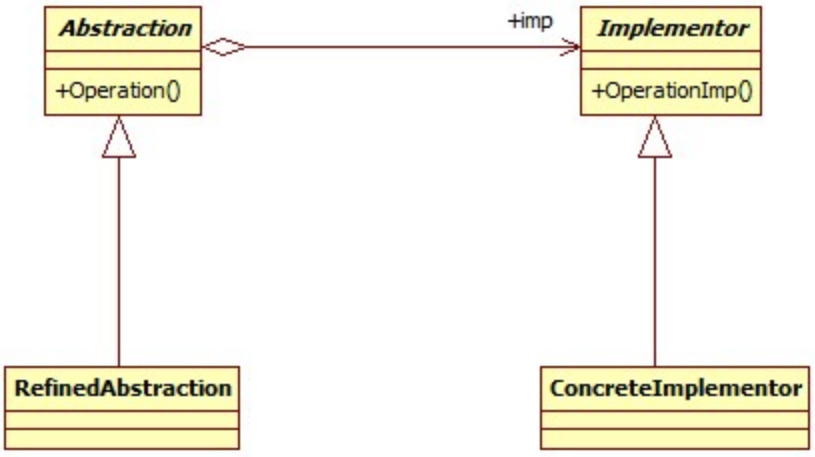

Figura 4.17 – Estrutura do *Bridge*

Na Fig.4.17 a classe *Abstraction* define a interface de abstração, mantendo uma referência a um objeto do tipo *Implementor*. A classe *RefinedAbstractrion* estende a interface definida por *Abstraction*. O *Implementor* define a interface para classes de implementação. Esta interface não tem a obrigação de corresponder exatamente à interface de abstração. Normalmente, a interface de implementação fornece apenas operações primitivas cabendo à classe *Abstraction* a responsabilidade de definir operações de alto nível baseadas nestas primitivas. A

classe *ConcreteImplementor* contém a implementação concreta da interface definida por *Implementor*.

Tabela 4.18 - Quando utilizar o *Bridge*

Aplicabilidade	• Para evitar a ligação permanente entre uma abstração e sua implementação; • Tanto a abstração quanto a implementação devem ser extensíveis através de herança; • Mudanças na implementação de uma abstração não devem ter impacto sobre o cliente.

4.2.5 Facade

O padrão de projeto *Facade* apresenta como objetivo oferecer uma interface única para um conjunto de interfaces de um subsistema. Ele define uma interface de nível mais elevado que torna o subsistema mais fácil de usar. Algumas vezes é preciso estruturar um sistema em subsistemas contribuindo para reduzir sua complexidade. A dependência entre subsistemas pode ser minimizada através do uso de um objeto Fachada, o qual provê uma interface única e uniforme para as diversas funcionalidades de um subsistema.

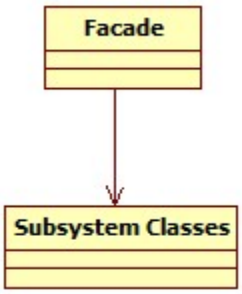

Figura 4.19 – Estrutura do *Facade*

Na Fig.4.19 a classe *Facade* tem o conhecimento de quais classes do subsistema seriam responsáveis pelo atendimento de uma solicitação. Delegando as solicitações de clientes a objetos apropriados dos subsistemas. As classes de subsistema implementam as funcionalidades do subsistema. Respondendo as solicitações de serviços do *Facade*. Essas classes não têm conhecimento da *Facade*.

Tabela 4.20 - Quando utilizar o *Facade*

Aplicabilidade	• For preciso prover uma interface simplificada para um subsistema complexo. Um *Facade* pode prover uma visão simples e *default* do subsistema, suficiente para a maioria dos clientes. • Existir muitas dependências entre clientes e classes da implementação. O *Facade* reduz esta dependência e promove independência e portabilidade. • Deseja-se criar sistemas em camadas. Um *Facade* provê o ponto de entrada para cada camada (nível) do subsistema.

4.2.6 Flyweight

O padrão de projeto *Flyweight* tem como objetivo usar o compartilhamento para dar suporte a vários objetos de forma eficiente. Algumas aplicações poderiam se beneficiar de sua estruturação em objetos em seu projeto, mas uma implementação mal elaborada seria proibitivamente cara.

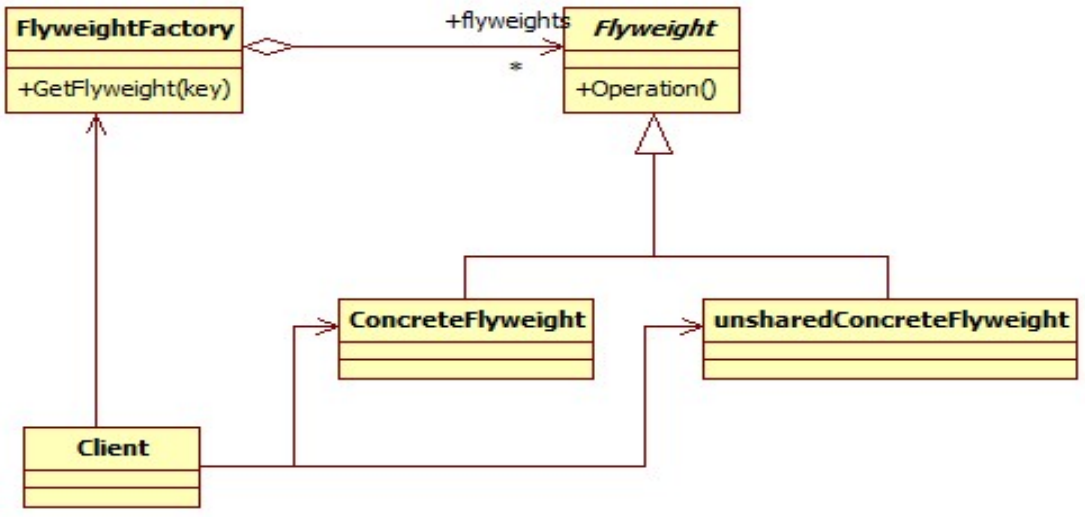

Figura 4.21 – Estrutura do *Flyweight*

Na Fig.4.21 a classe *Flyweight* define uma interface de entrada para os Flyweights. *ConcreteFlyweight* é responsável pela implementação da interface do *Flyweight* e adiciona estados intrínsecos se existirem. O *ConcreteFlyweight* precisa ser compartilhado e qualquer estado que ele armazene precisa ser intrínseco. Na classe *UnsharedConcreteFlyweight* nem todas as classes que implementam

Flyweight precisam ser compartilhadas. A interface possibilita compartilhamento, mas não obriga. O *FlyweightFactory* fabrica *Flyweight*. Quando o *client* pede um *flyweight*, só ira ser criado se ainda não tiver sido criado antes. Caso contrário retorna a referência que já existe. Finalmente *Client* armazena referencias para os objetos. Além de processar ou armazenar os estados extrínsecos do Flyweight.

Tabela 4.22 - Quando utilizar o *Flyweight*

Aplicabilidade	• Custos de armazenamento são altos, por causa da imensa quantidade de objetos; • Parte considerável do estado do objeto pode ser tornar extrínseco; • Uma vez que o estado extrínseco é removido, muitos agrupamentos de objetos podem ser substituídos por uma quantidade consideravelmente menor de objetos compartilhados; • A aplicação não depende de identidade de objetos.

4.2.7 Proxy

O padrão de projeto *Proxy* tem como objetivo fornecer um objeto representante ou marcador de outro objeto para controlar o acesso ao mesmo. Uma razão para controlar o acesso a um objeto é retardar o custo de sua criação e inicialização até o momento em que realmente é necessário utilizá-lo. Este retardamento torna-se necessário, pois muitas vezes, por exemplo, estes objetos podem ser recursos gráficos custosos de serem criados todos de uma vez, pois muitas vezes eles seram visualizados separadamente.

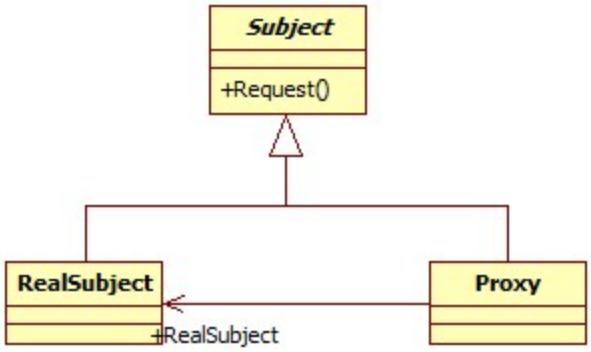

Figura 4.23 – Estrutura do *Proxy*

Na Fig.4.23 a classe *Subject* define uma interface comum para que um *Proxy* e um *RealSubject* tornem-se intercambiáveis. O *Proxy* mantém uma referência para acessar o *RealSubject*. Provendo uma interface idêntica ao *Subject* para permitir que este, substitua o *RealSubject*. Controlando também, o acesso ao *RealSubject*. O *RealSubject* é o objeto real que o *Proxy* representa.

Tabela 4.24 - Quando utilizar o *Proxy*

Aplicabilidade	• For preciso contar o numero de referencias para o objeto real; • For preciso carregar um objeto persistente para a memoria quando ele for referenciado pela primeira vez; • For preciso verificar se o objeto real esta bloqueado antes de ser acessado, para certificar-se que outro objeto não o alterou.

4.3 Padrões comportamentais

Os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidades entre objetos. Estes padrões descrevem também padrões de comunicação entre objetos ou classes. Os padrões comportamentais de classe utilizam a herança para distribuir o comportamento entre classes. Já os de objetos, utilizam a composição de objetos. Segundo Lozano (2003) “Os Padrões comportamentais explicitam a forma como os componentes da aplicação se comunicam para atingir um objetivo”.

4.3.1 Observer

O padrão de projeto *Observer* tem como finalidade definir uma relação de dependência 1:N de forma que quando o estado de um objeto e alterado, todos os estados dependentes deste, são notificados e atualizados automaticamente, possibilitando um baixo acoplamento entre os objetos e seus estados dependentes.

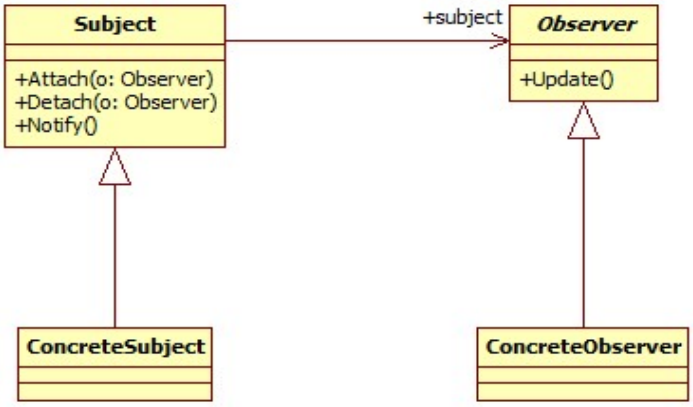

Figura 4.25 – Estrutura do *Observer*

Na Fig. 4.25 o *Subject* fornece uma interface para acoplar e desacoplar objetos *Observer*, e tem conhecimento de seu *Observer*. Qualquer número de objetos *Observer* podem observar um *Subject*. O *ConcreteSubject* armazena o estado de interesse para *ConcreteObserver* e quando seu estado é modificado, ele envia uma notificação para seu(s) *Observer(s)*. A classe *Observer* define uma interface de atualização para objetos que devem ser notificados sobre mudanças em um *Subject*. No *ConcreteObserver* são mantidas uma referência para um objeto *ConcreteSubject* e armazenado o estado que deve ficar consistente com o de *Subject*. Além de ser implementado o *Observer*, atualizando a interface para manter seu estado consistente com o de Subject.

Tabela 4.26 - Quando utilizar o *Observer*

Aplicabilidade	• Quando uma abstração apresenta dois aspectos, um dependente do outro. Encapsulando estes aspectos em objetos separados permite que os varie e reutilize de forma independente. • Quando uma modificação em um objeto requer modificação em outros, e não se sabe (em tempo de programação) quantos objetos precisam ser modificados. • Quando um objeto deve ser apto a notificar outros objetos sem saber quem são estes objetos. Em outras palavras, quando os quer fracamente acoplados.

4.3.2 Strategy

O padrão de projeto *Strategy* tem como objetivo permitir que uma família de algoritmos seja utilizada de modo independente e seletivo. Este padrão defini uma família de algoritmos e encapsula cada um deles, tornando-os intercambiáveis. O *Strategy* permite mudar os algoritmos independentemente dos clientes que os usam.

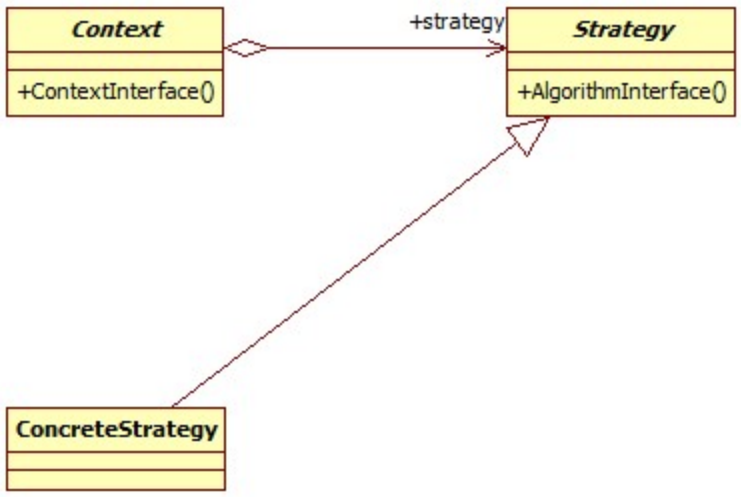

Figura 4.27 – Estrutura do *Strategy*

Na Fig.4.27 a classe abstrata *Strategy* é responsável pela fatoração do código comum entre os algoritmos. A classe *ConcreteStrategy* é responsável pela implementação do algoritmo. E o *Context* é configurado com um objeto *ConcreteStrategy* e mantém uma referência para um objeto *Strategy*, além de poder definir uma interface para que o *Strategy* acesse seus dados.

Tabela 4.28 - Quando utilizar o *Strategy*

Aplicabilidade	• Muitas classes relacionadas diferem apenas em seus comportamentos. *Strategies* provêem uma maneira de configurar uma classe com um entre vários comportamentos possíveis; • É necessária diferente variação de um algoritmo. *Strategies* podem ser usados quando essas variações são implementadas como uma classe hierárquica de algoritmos;

4.3.3 Template method

O padrão de projeto *Template Method* tem finalidade de definir o esqueleto de um algoritmo, permitindo que subclasses customizem o restante do algoritmo. Assim, permitindo que subclasses redefinam certos aspectos de um algoritmo sem modificar a estrutura do algoritmo.

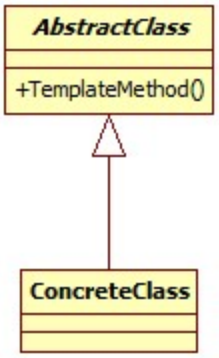

Figura 4.29 – Estrutura do *Template Method*

Na Fig.4.29 o padrão *Template Method* possui a classe *AbstractClass* que define as operações abstratas, que as subclasses concretas definem para implementar certas etapas do algoritmo. Este padrão implementa um método *Template Method* definindo o esqueleto de um algoritmo, onde este método chama várias operações, entre as quais as operações abstratas da classe. No *ConcreteClass* é implementado as operações abstratas, que serão responsáveis pelo comportamento específico desta subclasse.

Tabela 4.30 - Quando utilizar o *Template Method*

Aplicabilidade	• É desejado implementar partes invariáveis de um algoritmo e deixar que as subclasses implementem os comportamentos variáveis; • Comportamentos comuns entre subclasses devem ser fatorados e localizados em uma classe comum evitando duplicação de código.

4.3.4 Chain of Responsibility

O padrão de projeto *Chain of Responsibility* tem como objetivo evitar dependência do remetente (cliente) de uma requisição ao seu destinatário. Encadeia

os objetos atendentes e passa a solicitação através desta cadeia até que algum deles a trate.

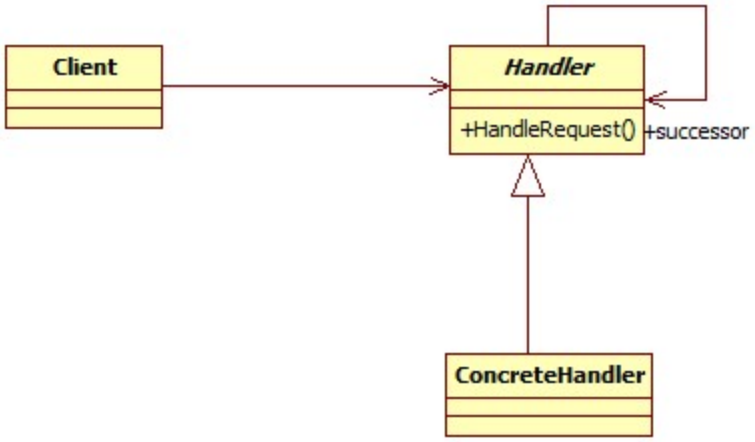

Figura 4.31 – Estrutura do Chain of Responsibility

Na Fig.4.31 a classe *Handler* define a interface para tratar as solicitações, implementando a referência ao sucessor. O *ConcreteHandler* trata as solicitações pelas quais é responsável e pode acessar seu sucessor. Caso *ConcreteHandler* não consiga tratar a solicitação ele repassa a solicitação para o seu sucessor. O *Client* inicia a solicitação para um objeto *ConcreteHandler* da cadeia.

Tabela 4.32 - Quando utilizar o *Chain of Responsibility*

Aplicabilidade	• Mais de um objeto pode tratar de um pedido, e o tratador de pedidos (*handler*) não é logo conhecido. O *handler* deve ser buscado automaticamente de forma ascendente. • Quer-se emitir um pedido para um de vários objetos sem especificar o recebedor de forma explícita. • O conjunto de objetos que pode tratar de um pedido deve ser configurado dinamicamente.

4.3.5 Command

O padrão de projeto *Command* possui como finalidade associar uma ação a diferentes objetos através de uma interface conhecida. Ele, encapsula uma solicitação como um objeto, assim, permitindo que se parametrize clientes com diferentes solicitações, filas ou registros de solicitações, suportando ainda o cancelamento de solicitações.

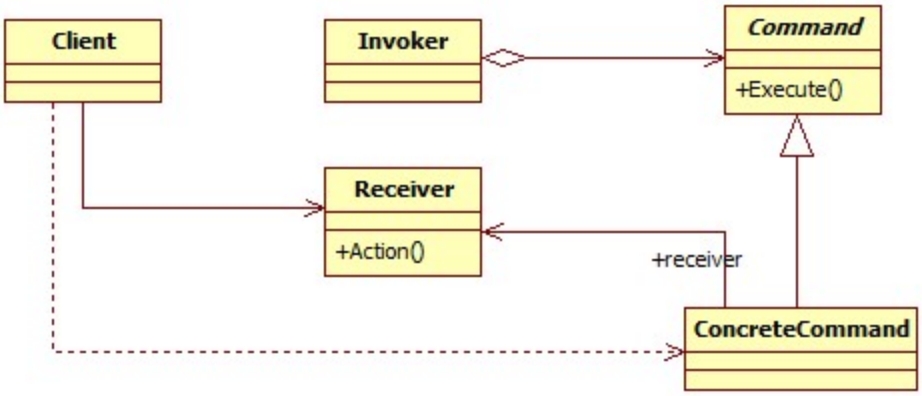

Figura 4.33 – Estrutura do *Command*

Na Fig.4.33 a classe *Command* faz a declaração da interface para a execução de uma operação. O *ConcreteCommand* faz a definição de uma vinculação entre um objeto *Receiver* e uma ação. Ele faz a implementação do método *execute* através da invocação da(s) correspondente(s) operação(ções) no *Receiver*. A classe *Client* faz a criação de um objeto *ConcreteCommand* e estabelece o seu receptor *(Receiver)*. O *Invoker* faz a solicitação ao *Command* para execução da solicitação. E o *Receiver* tem conhecimento de como executar as operações associadas a uma solicitação.

Tabela 4.34 - Quando utilizar o *Command*

Aplicabilidade	• Deseja-se parametrizar objetos por uma ação a ser executada; • Deseja-se especificar, enfileirar ou executar requisições em diferentes momentos. Um objeto *Command* tem um tempo de vida independente da requisição original.

4.3.6 Interpreter

O padrão de projeto *Interpreter* tem como objetivo ajudar uma aplicação a entender uma declaração de linguagem natural e executar a funcionalidade da declaração. Dada uma linguagem, o padrão define uma representação para a gramática da linguagem, juntamente com um interpretador que utilizar esta representação para interpretar sentenças na linguagem.

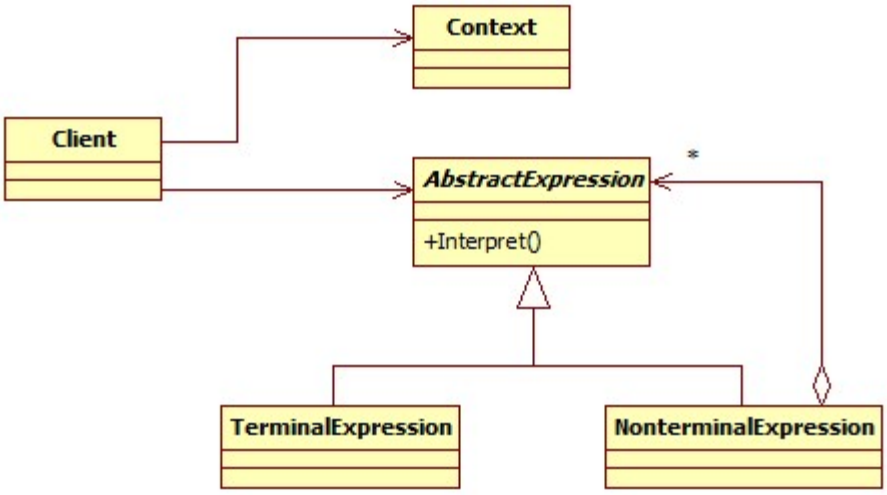

Figura 4.35 – Estrutura do *Interpreter*

Na Fig.4.35 o *AbstractExpression* faz a implementacao da interface comum a todos os nós da árvore sintática. O *TerminalExpression* faz a implementação das operações de interpretação para os símbolos terminais e o *NonterminalExpression* faz a implementação das operações de interpretação para os símbolos não-terminais. A operação de interpretação é chamada recursivamente e a base da recursão é um símbolo terminal. No *Context* estão contidas as informações que são globais ao interpretador, tais como o valor das variáveis para uma instância do problema e o resultado das operações realizadas. Finalmente, o *Client* constrói a árvore sintática abstrata a partir de instâncias de nós terminais e não-terminais.

Tabela 4.36 - Quando utilizar o *Interpreter*

Aplicabilidade	• A gramática é simples; • A eficiência não é uma preocupação crítica.

4.3.7 Iterator

O padrão de projeto *iterator* tem como finalidade prover uma maneira de percorrer seqüencialmente os elementos de uma coleção sem violar o seu encapsulamento.

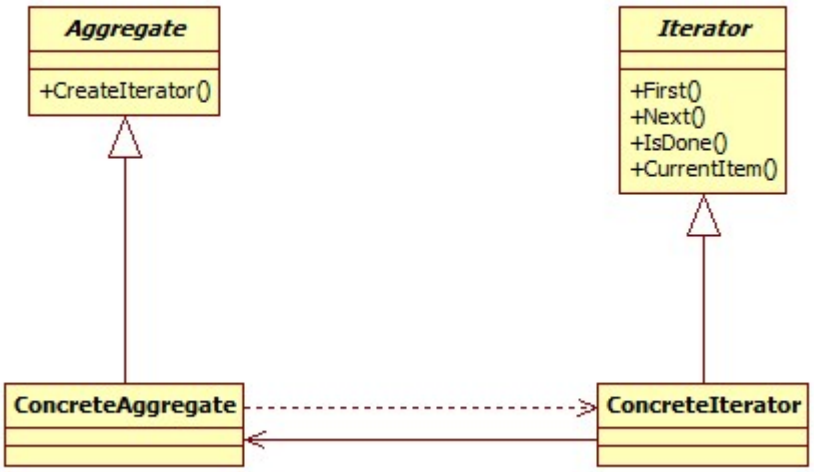

Figura 4.37 – Estrutura do *Iterator*

Na Fig.4.37 a classe *Iterator* tem responsabilidade de definir a interface para acessar e percorrer os elementos. O *ConcreteIterador* faz a implementação da interface Iterator mantendo o controle da posição corrente no percurso do *Aggregate*. O *Aggregate* faz a definição de interface para a criação de um objeto *iterador*. E o *ConcreteAggregate* realiza a implementação da interface de criação do Iterador para retornar uma instância do *ConcreteIterador*.

Tabela 4.38 - Quando utilizar o *Iterator*

Aplicabilidade	• Deseja-se acessar o conteúdo de um objeto agregado sem expor sua representação interna. • Deseja-se suportar múltiplas travessias de um objeto agregado. • Quer-se prover uma interface uniforme para atravessar diferentes estruturas agregadas. Suportar iteração polimórfica.

4.3.8 Mediator

O padrão de projeto *Mediator* tem como finalidade criar um objeto para agir como um mediador controlando a interação entre um conjunto de objetos. Promovendo, um acoplamento fraco entre objetos, a fim de evitar a referencia direta o um outro objeto, permitindo que se possa variar a interação entre eles de modo independente.

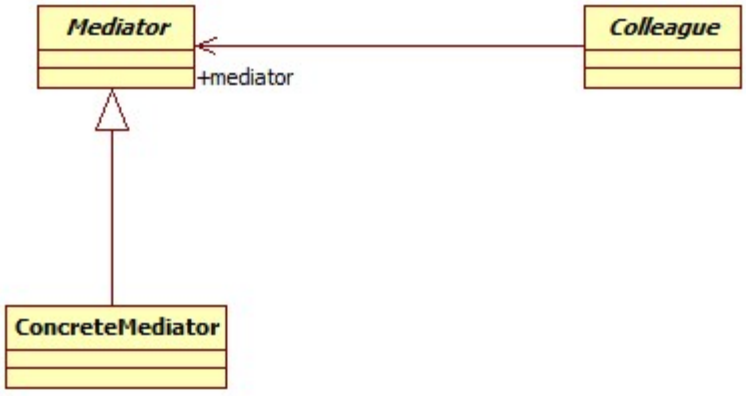

Figura 4.39 – Estrutura do *Mediator*

Na Fig.4.39 a classe Mediator faz a definição da interface para a comunicação com objetos Colleague. O ConcreteMediator faz a implementação do comportamento cooperativo pela coordenação de objetos Colleague, além de conhecer e manter seus colleagues. Cada classe Colleague possui conhecimento de seu objeto Mediator. Cada objeto Colleague se comunica com seu mediador sempre que o mesmo necessite comunicar-se com outro objeto Colleague. Os objetos apenas conhecem o Mediator, assim, diminuindo o número de interconexões.

Tabela 4.40 - Quando utilizar o Mediator

Aplicabilidade	• Um conjunto de objetos se comunica de uma forma bem definida, porém complexa. As interdependências resultantes não são estruturadas e são de difícil compreensão; • O reuso de objetos é difícil uma vez que um objeto se refere e se comunica com muitos outros objetos; • Um comportamento que é distribuído entre várias classes deve ser customizado sem o uso excessivo de sub-classes.

4.3.9 Memento

O padrão de projeto *Memento* tem como objetivo tornar possível salvar o estado de um objeto de modo que o mesmo possa ser restaurado. Sem violar a encapsulação, ele captura e armazena externamente o estado de um objeto.

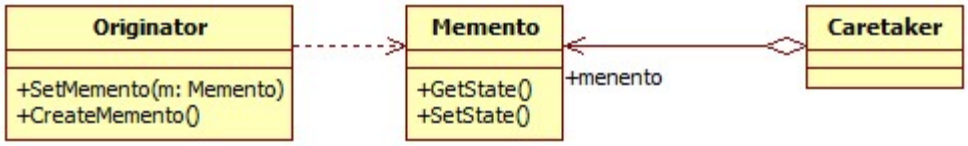

Figura 4.41 – Quando utilizar o *Memento*

Na Fig.4.41 a classe *Originator* é responsável pela criação de um objeto *Memento* contendo informações de seu estado atual. O *Originator* também utiliza o *memento* para restaurar seu estado interno. O *Memento* guarda o estado interno do objeto *originator*. E o *Caretaker* guarda os *mementos* do *originator.*

Tabela 4.42 - Quando utilizar o *Memento*

Aplicabilidade	• O estado instantâneo de um objeto deve ser salvo, até que ele possa ser restaurado para aquele estado mais tarde. • Uma interface direta para obter o estado do objeto exporia detalhes de implementação, quebrando assim o encapsulamento do objeto.

4.3.10 State

O padrão de projeto *State* tem como finalidade permitir que um objeto possa alterar seu comportamento quando o seu estado interno mudar. O objeto parecera ter mudado sua classe.

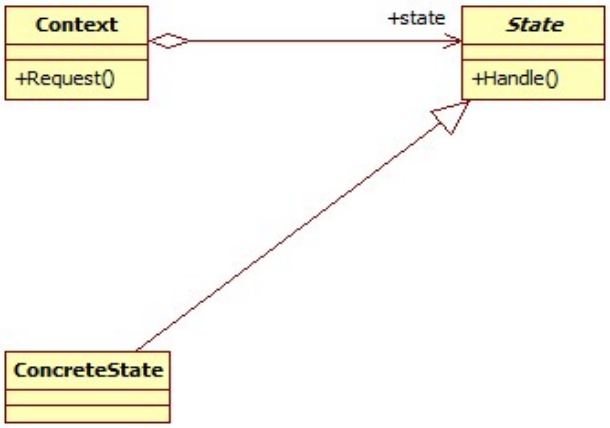

Figura 4.43 – Estrutura do *State*

Na Fig.4.43 a classe *Context* define a interface de interesse para os clientes. Ela mantém uma instância de uma subclasse *ConcreteState* que é responsável por

definir o estado corrente. O *State* faz a implementação da interface para encapsular o comportamento associado com um estado particular do contexto. E o *ConcreteState* é responsável por implementar o comportamento associado a um estado do Context.

Tabela 4.44 - Quando utilizar o *State*

Aplicabilidade	• O comportamento de um objeto depende de seu estado, e este deve ser mudado em tempo de execução conforme as mudanças ocorridas em seu estado. • Operações que possuem comandos condicionais muito grandes, que dependem do estado do objeto. Este estado é usualmente representado por uma ou mais constantes enumeradas. Freqüentemente, muitas operações irão conter a mesma estrutura condicional. O State coloca cada ramo dessa estrutura em uma classe separada. Dessa maneira, o estado do objeto pode ser tratado como um objeto com seus próprios direitos que podem variar independentemente dos outros objetos.

4.3.11 Visitor

O padrão de projeto *Visitor* tem como objetivo definir operações independentes a serem realizadas sobre elementos de uma estrutura. Ele representa uma operação a ser executada sobre os elementos da estrutura de um objeto. Este padrão possibilita a definição de novas operações sem modificar as classes dos elementos sobre as quais ele atua.

Na Fig.4.45 a classe *Visitor* realiza a declaração de uma operação de visita para cada classe de *ConcretElement* na estrutura de objetos. O nome da operação identifica a classe que chamadora do *Visitor*. Sendo assim, o *Visitor* tem conhecimento da classe concreta do elemento sendo visitado e pode acessar este objeto por sua interface. O *ConcreteVisitor* faz a implementação de cada operação declarada pelo *Visitor*. Onde, cada operação é responsável por implementar um fragmento do algoritmo definido para a classe correspondente de objetos na estrutura. O *ConcreteVisitor* pode acumular estados durante a varredura da estrutura de objetos. No *Element* é definido o método *accept* que recebe um *Visitor*. Na classe *ConcreteElement* é implementado a operação *accept* que recebe um *Visitor* e chama

a operação de visita apropriada deste *Visitor*. Finalmente o *ObjectStructure* pode enumerar seus elementos, podendo prover uma interface de mais alto nível para que o *Visitor* visite os elementos.

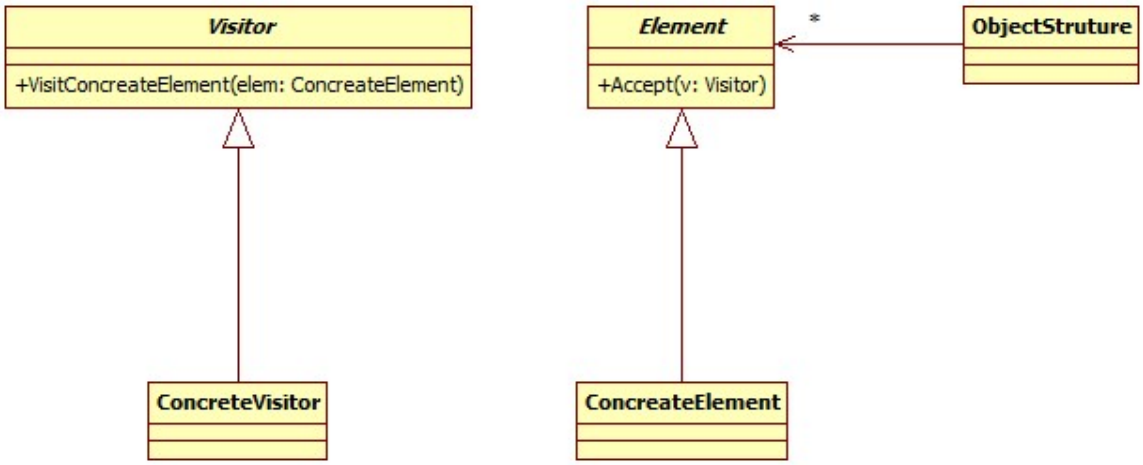

Figura 4.45 - Estrutura do *visitor*

Tabela 4.46 - Quando utilizar o *Visitor*

Aplicabilidade	• Uma estrutura de objetos contém várias classes de objetos com diferentes interfaces, e quer-se realizar operações nesses objetos que dependem de suas classes concretas.; • Muitas operações distintas e não relacionadas precisam ser feitas em objetos numa mesma estrutura de objetos, e se quer evitar uma "poluição" dessas classes com essas operações. O Visitor permite que se guardem operações relacionadas juntas em uma classe. Quando uma estrutura de objetos é compartilhada por várias aplicações, usa-se o Visitor para capturar operações apenas para aquelas aplicações que necessitem delas. • As classes definidas em uma estrutura de objetos raramente mudam, mas freqüentemente se quer definir novas operações sobre essa estrutura.

5 Pesquisa de campo

5.1 Introdução

O presente relatório apresenta os resultados obtidos pela pesquisa de campo realizada em algumas empresas de desenvolvimento Web na cidade de Pelotas, tendo por objetivo analisar o atual estágio de utilização de padrões de projeto no desenvolvimento de aplicações em Ambiente Web nessas empresas. A pesquisa apresenta-se, ainda, como instrumento fundamental para a seleção dos padrões de projeto abordados no capítulo 6 deste trabalho.

5.2 Objetivos

Esta pesquisa apresenta os seguintes objetivos:

- Avaliar a utilização de padrões de projeto;
- Aporte para definição dos padrões de projeto que serão foco de estudo no capitulo seguinte;
- Confrontar os padrões mais utilizados nas empresas investigadas com os padrões ditos mais populares pela literatura;
- Verificar quais os padrões de projeto são mais utilizados;
- Verificar quais os tipos de padrões são mais utilizados;
- Verificar qual a linguagem de programação mais utilizada no desenvolvimento Web na cidade;
- Verificar a utilização de *frameworks* no desenvolvimento Web.

5.3 Metodologia

A metodologia de trabalho se desenvolveu com a utilização da técnica de pesquisa quantitativa. Primeiramente foi construído o instrumento de coleta de dados, sendo este um questionário fechado composto por perguntas objetivas e de marcar, o questionário utilizado pode ser visto no apêndice A. Na elaboração deste questionário, foram pré-selecionados um conjunto de oito padrões de projeto definidos como os padrões mais populares por Gamma (2000), com a intenção de

confrontar a literatura com a realidade da cidade de Pelotas no que diz respeito à utilização de padrões de projeto no desenvolvimento de aplicações em ambiente Web. Posteriormente foi definida a amostra, para isso foram realizados contatos com empresas do ramo de desenvolvimento Web da cidade de Pelotas, totalizando 6 empresas participantes. Cada responsável pela empresa indicou um desenvolvedor chefe para participar da pesquisa e também assinou o consenso de pesquisa autorizando a publicação dos dados coletados. Estes documentos podem ser vistos no apêndice B.

A pesquisa deu-se em dois momentos, no primeiro momento foi realizada uma entrevista com desenvolvedor indicado, onde foram realizadas perguntas básicas seguindo o roteiro do questionário, com duração média de 20 minutos. Já no segundo momento, na qual o objetivo era identificar os padrões de projeto utilizados pela empresa, o processo foi mais demorado e se deu da seguinte forma: primeiramente o entrevistado era questionado se tinha conhecimento sobre padrões de projeto e se esses eram utilizados por sua equipe de desenvolvimento. Caso o entrevistado afirmasse sim, então esses eram contabilizados e essa empresa marcada como uma daquelas que utilizam padrões de projeto. Quando o entrevistado afirmava que padrões de projeto não eram utilizados ou não sabia se eram utilizados, então, o entrevistador questionava o entrevistado sobre de que forma, sua equipe de desenvolvimento, resolvia alguns problemas de implementação presentes freqüentemente em todos os sistemas. Através da analise dessa resposta e também de alguns códigos fontes, desenvolvidos pela equipe do entrevistado, relativos à solução daqueles problemas, era verificado se a solução construída se tratava de uma metodologia de solução proposta por alguns dos padrões de projeto selecionados, sendo assim era feita a contabilização do padrão e essa empresa marcada como uma daquelas que utilizam padrões sem conhecimento prévio ou, caso contrário, como uma daquelas que não utilizam padrões de projeto. Todo esse processo tinha um tempo de duração médio de uma semana.

5.4 Amostra

A amostra para esta pesquisa foi composta por 6 empresas do ramo de desenvolvimento Web da cidade de Pelotas, as empresas participantes foram as seguintes: Checkplant Sistemas de Rastreabilidade, Conrad Caine Media

Applications, REWEB, Coinpel, ADSWeb e Centro de Informática. A descrição completa de cada empresa pode ser vista no apêndice C.

5.5 Resultados

Nesta seção são apresentados os resultados da pesquisa de campo através de tabelas e gráficos.

5.5.1 Utilização de padrões de projeto

O percentual das empresas que utilizam padrões de projeto, na cidade de Pelotas (Fig.5.2), para o desenvolvimento de aplicações Web abrange a maioria do universo pesquisado. As empresas que utilizam esse recurso representam 66%, já as empresas que não utilizam corresponde a 17% e as que não sabem que utilizam este tipo de recurso compreendem 17%.

Utilização de Padrões de projeto

Empresa	É Utilizado	Não sabem que utilizam	Não é utilizado
Checkplant	X		
Conrad Caine	X		
ReWeb	X		
ADSWeb		X	
Coinpel			X
Centro de Informática	X		
Total	4	1	1

Figura 5.1 – Padrões de projeto utilizados por cada empresa

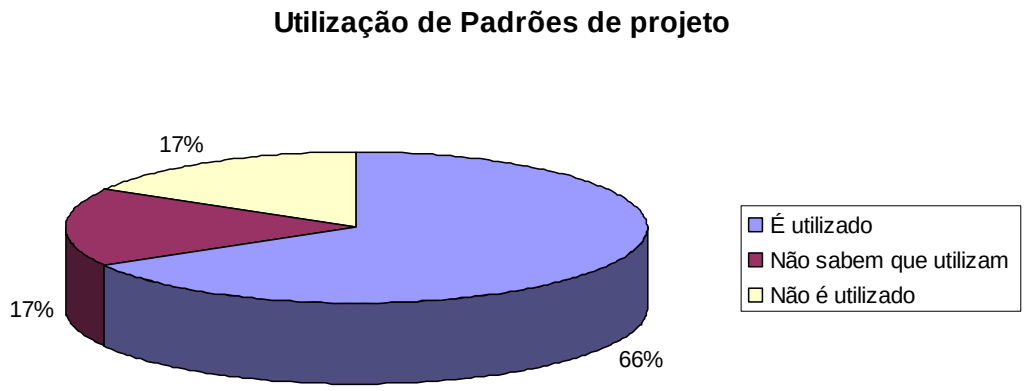

Figura 5.2 – Utilização de Padrões de projeto

5.5.2 Padrões de projeto mais utilizados

Quando analisado o universo investigado em relação aos padrões de projeto mais utilizados (Fig.5.3), é possível verificar que os padrões de projeto *Factory Method* 17% e *Adapter* 23% apresentam uma superioridade em termos de utilização para desenvolvimento Web em relação aos outros padrões. Outra importante constatação possível, é que o grupo de padrões de projeto mais utilizados no desenvolvimento de aplicações em ambiente Web (Fig.5.3), nas empresas investigadas, vai ao encontro dos padrões de projeto ditos mais populares pela literatura (GAMMA, 2000). Este grupo, identificado pela pesquisa de campo, será foco de estudo no capítulo 6 deste trabalho, onde para cada padrão será apresentado um estudo de caso de aplicação desse padrão no desenvolvimento de aplicações Web.

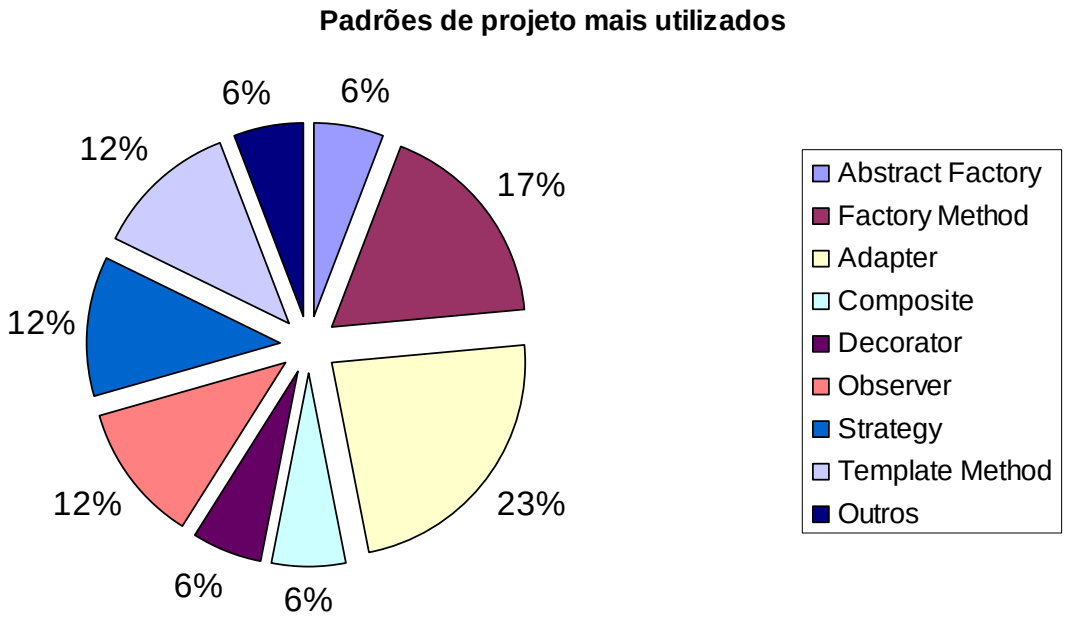

Figura 5.3 – Padrões de projeto mais utilizados

5.5.3 Empresas x Padrões de Projeto

A investigação sobre quais padrões de projeto cada empresa participante da pesquisa utiliza pode ser vista na Fig.5.4:

Ëmpresas	Abstract Factory.	Factory Method	Adapter	Composite	Decorator	Observer	Strategy	Template Method	Outros
Checkplant		X	X		X	X			
Conrad Caine		X	X	X		X	X	X	X
ReWeb		X	X					X	
ADSWeb	X						X		
Coinpel									
Centro de Informática			X						

Figura 5.4 – Empresas x Padrões de projeto

Os resultados do estudo demonstram que a empresa Conrad Caine é a maior utilizadora de padrões de projeto dentre as empresas investigadas (Fig.5.5), nesta empresa foram identificados 7 padrões de projeto. Analisando as características dessa empresa, descritas no apêndice C, entre outras é verificado que esta empresa desenvolve software de elevada qualidade. A partir disto pode-se supor que existe uma relação de proporcionalidade entre a qualidade de um software e a utilização de padrões de projeto. Também foi observado que a Conrad Caine foi à única das empresas pesquisadas que utiliza outros padrões de projeto além dos padrões descritos na Fig.5.3.

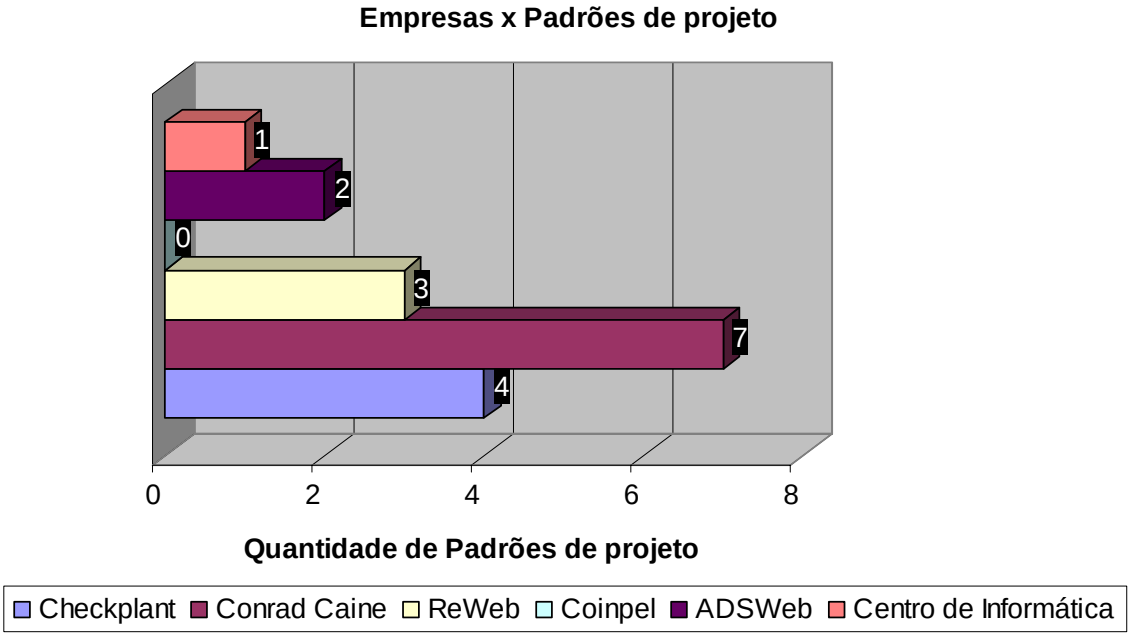

Figura 5.5 – Quantidade de padrões de projeto utilizados por empresa

Observou-se também que existem empresas que utilizam padrões de projeto sem um conhecimento prévio. Na empresa ADSWeb foi identificado a utilização de dois padrões de projeto o *Abstract Factory* e o *Strategy*, porém seu grupo de desenvolvimento não tinha o conhecimento de que utilizavam padrões de projeto, isso só veio a acontecer a partir da participação nesta pesquisa, onde por intermédio de discussões, durante a fase de entrevista, entre o grupo de desenvolvedores e o entrevistador foi verificado a utilização daqueles padrões.

5.5.4 Tipos de padrões de projeto mais utilizados

Quando analisado o universo pesquisado em relação aos tipos de padrões mais utilizados (Fig.5.6), os tipos mais utilizados são os padrões de projeto Comportamentais 40%, em segundo os padrões do tipo Estruturais 33% e 27% os do tipo de Criação. A partir disto tem-se um indicativo de que as empresas investigadas apresentam uma preocupação maior com a forma como os componentes de uma aplicação se comunicam para atingir um objetivo, visto que essa é a principal característica dos padrões comportamentais.

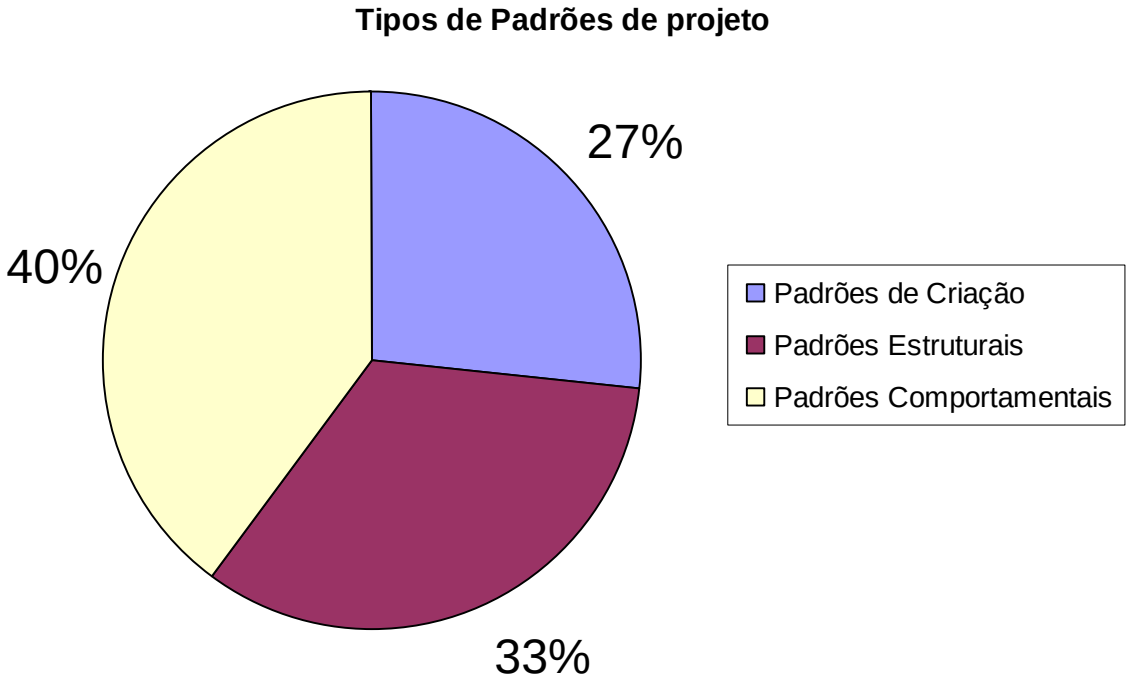

Figura 5.6 – Tipos de Padrões de projeto mais utilizados

5.5.5 Linguagens de programação mais utilizadas

Os resultados da pesquisa em relação às linguagens de programação mais utilizadas para desenvolvimento Web podem ser vistos na Fig.5.7, onde foi apontada a linguagem PHP como sendo mais utilizada, visto que 100% das empresas investigadas utilizam esta linguagem.

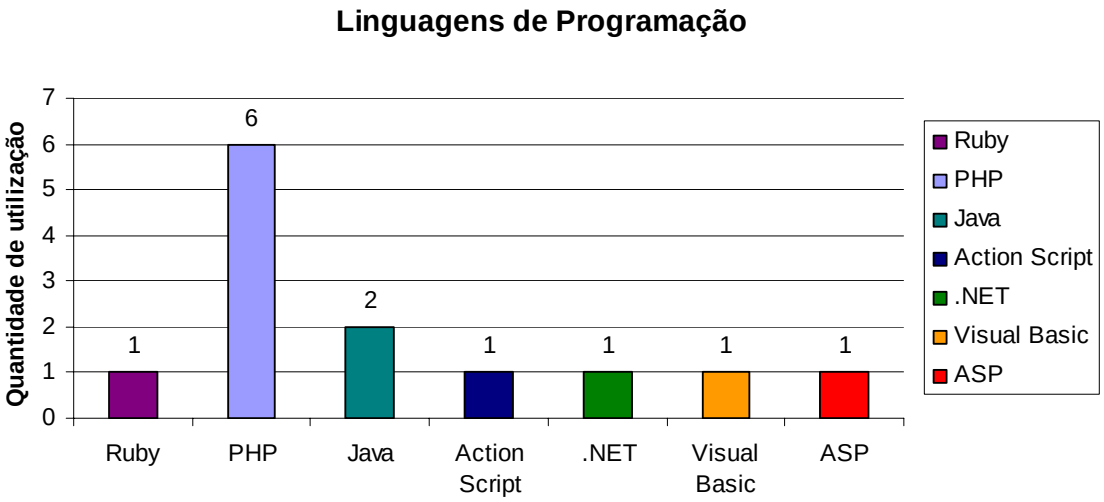

Figura 5.7 – Linguagens de programação mais utilizadas

5.5.6 *Frameworks* para desenvolvimento mais utilizados

Quando analisado o universo pesquisado em relação aos *frameworks* mais utilizados para o desenvolvimento de aplicações Web (Fig.5.8), é verificado que a grande maioria das empresas investigadas (83%) utiliza um *framework* próprio para o desenvolvimento de suas aplicações ou uma combinação de um *framework* do mercado com um próprio. A justificativa para isto, esta relacionada ao problema de um *framework* do mercado não atender todas as necessidades das empresas investigadas no que diz respeito ao processo de desenvolvimento de software.

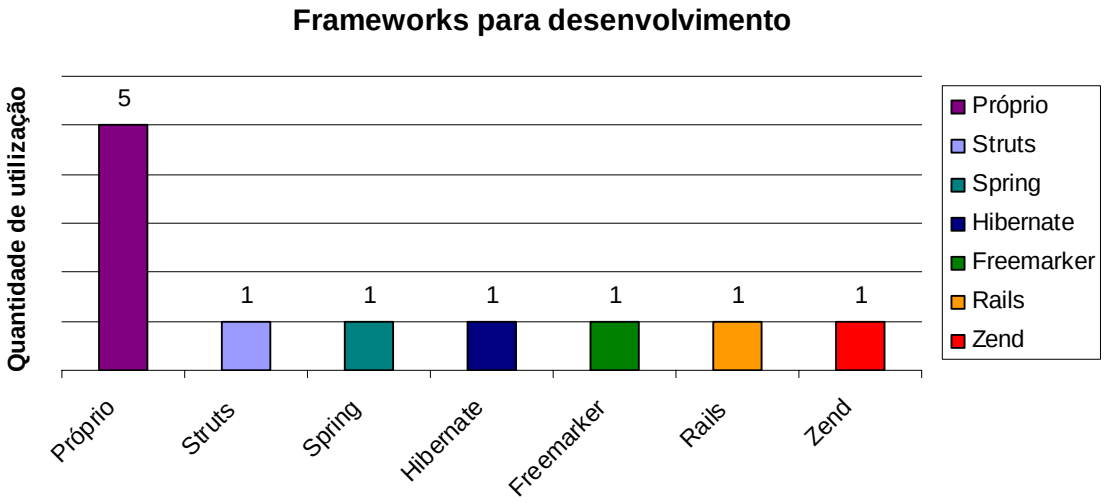

Figura 5.8 – *Frameworks* para desenvolvimento

5.5.7 Considerações

Através desta pesquisa foi possível mostrar o atual estágio de utilização de padrões de projeto, no desenvolvimento de aplicações em ambiente Web, em algumas empresas na cidade de Pelotas. Esta pesquisa não teve nenhum objetivo de comparação entre as empresas participantes da mesma, mais sim focar na análise e verificação quanto à utilização de padrões de projeto.

Também é importante considerar, que as empresas participantes da pesquisa na pessoa de seus responsáveis, apresentaram um grande interesse nos resultados finais deste trabalho, pois segundo eles, existe um desejo comum de saber que práticas e metodologias estão sendo adotadas pela comunidade de desenvolvimento de software.

6 Estudos de caso de aplicação dos padrões de projeto

Neste capítulo são realizados estudos de caso de aplicabilidade, no desenvolvimento de aplicações em ambiente Web, de cada um dos padrões de projeto que foram selecionados através da pesquisa de campo apresentada no capítulo 5, mais especificadamente na Fig.5.3. Para mostrar como os padrões de projeto são aplicados na prática, é utilizada a linguagem de programação PHP.

6.1 Estudo de caso do *Factory Method*

No estudo de caso de utilização do padrão de projeto *Factory Method* no desenvolvimento de aplicações em ambiente web, é apresentada uma solução abordada por Casanova (2008 b), para dinamizar a maneira como os *requires/includes*, das classes que compõem uma aplicação, são realizados.

A solução para este problema de projeto, apresentada a seguir, possibilita uma diminuição significativa no número de *requires/includes* que geralmente é feito, em uma aplicação que não utiliza este padrão. A grande vantagem dessa solução, utilizando o *Factory Method*, é que o *require/include* de uma classe, que compõem a aplicação, é realizado de forma dinâmica, somente quando for realmente necessário uma instancia dessa classe, dispensado a realização do *require/include* de todas as classes da aplicação no cabeçalho do arquivo index da aplicação.

Assim sendo, esta solução possibilita um ganho de desempenho de execução da aplicação. Pois quanto maior for o número de *requires/includes* que uma aplicação realiza, em um mesmo corpo de execução, maior a quantidade de arquivos que serão vinculados a este corpo. Conseqüentemente, tornando o desempenho da aplicação, como um todo, mais lento. Na Fig. 6.1 é mostrado um exemplo de código **sem** utilizar *o Factory Method*.

```php
<?
//c:/development/project/index.php

    //classes da aplicação
    include("classeA.php");
    include("classeB.php");
    include("classeC.php");

    // ...

    //Sessão da aplicação a ser carregada
    include("teste.php");
?>
```

Figura 6.1 – Código fonte do arquivo index.php sem utilizar o *Factory Method*

```php
<?
//c:/development/project/teste.php

    //Instaciando objetos necessários desta sessão
    $objeto1 = new ClasseA();
    $objeto2 = new ClasseB();
?>
```

Figura 6.2 – Código fonte do arquivo teste.php sem utilizar o *Factory Method*

Na Fig.6.1, nas linhas 5,6,7 é realizado includes de todas as classes que compõem uma aplicação. Na linha 12 da Fig.6.1 é realizado o include da sessão teste.php a ser executada. Porém, esta sessão só utiliza as classes A e B da aplicação, conforme linhas 5 e 6 da Fig.6.2. Então para esta sessão teste.php o include da classeC.php Fig.6.1 linha 7 é desnecessário. Na Fig. 6.3 é apresentada a solução para este problema **utilizando** o *Factory Method:*

```php
<?
//c:/development/project/index.php

    include("classeFabrica.php");

    //Sessão da aplicação a ser carregada
    include("teste.php");
?>
```

Figura 6.3 – Código fonte do arquivo index.php utilizando o *Factory Method*

```php
<?
//c:/development/project/teste.php

    //Utilizando o Factory Method
    $fabrica  = new Fabrica();

    //Instaciando objetos necessários desta sessão
    $objeto1 = $fabrica->getObjetoClasseA();
    $objeto2 = $fabrica->getObjetoClasseB();
?>
```

Figura 6.4 – Código fonte do arquivo teste.php utilizando *o Factory Method*

```php
<?
//c:/development/project/classeFabrica.php
    //Classe fábrica (Factory Method)
    class classeFabrica{
        public function __construct(){

        }
        //Métodos responsáveis por instaciar
        //objetos das classes da aplicação
        public function getObjetoClasseA(){
            require_once("classeA.php");
            return $objeto1 = new ClasseA();
        }
        public function getObjetoClasseB(){
            require_once("classeB.php");
            return $objeto2 = new ClasseB();
        }
        public function getObjetoClasseC(){
            require_once("classeC.php");
            return $objeto3 = new ClasseC();
        }
    }
?>
```

Figura 6.5 – Código fonte da Classe classeFabrica

Na Fig.6.3 linha 4 é realizado somente o include da classeFabrica,a qual é apresentada na Fig.6.5. Essa classe é responsável pela camada na qual ira interagir com todos os objetos da aplicação, sendo assim, não é preciso realizar includes de todas as classes da aplicação de uma vez só, conforme Fig.6.1 linhas 5, 6 e 7. Na Fig.6.4 linhas 8 e 9 há uma chamada para alguns dos métodos da classeFabrica. Esses métodos são responsáveis por requerir classes e instanciar objetos dinamicamente, conforme o padrão de projeto Factory Method propõem.

6.2 Estudo de caso do *Abstract Factory*

No estudo de caso do padrão de projeto *Abstract Factory*, é apresentado uma solução, também abordada por Casanova (2008 a), para o problema de necessidade de conexão com dois ou mais banco de dados e uma aplicação.

Na solução apresentada a seguir, é implementado a necessidade de conexão de uma aplicação em um banco de dados MySQL e também em um banco de dados PostgreSQL. Primeiramente, é descrito o problema de projeto e posteriormente sua solução. Para criar a conexão com cada um destes bancos de dados, é necessário um conjunto de diretivas, tais como URL, usuário, senha e outros, como mostrado na Fig.6.6 e Fig.6.7.

```
1 <?
2     $conexao = mysql_pconnect("localhost","root","senha");
3 ?>
```

Figura 6.6 – Código fonte para conexão com o bando de dados MySQL

```
1 <?
2     $conexao = pg_connect("localhost","root","senha");
3 ?>
```

Figura 6.7 – Código fonte para conexão com o bando de dados PostgreSQL

Então, é neste momento que aparece o primeiro problema, pois seria necessário repetir ao longo do código fonte da aplicação a linha de código contida na Fig.6.6 toda vez que houvesse necessidade de uma conexão com o banco de dados MySQL, ou a linha de código contida na Fig.6.7 de quando houvesse necessidade de conexão PostgreSQL. Outro problema surge, quando for necessário alterar o usuário e/ou senha de acesso a um dos bancos de dados, pois, para isso, seria necessário percorrer todos os arquivos de código, responsáveis por conexões ao banco de dados que se deseja alterar, para então se editar o código referente a usuário e/ou senha, gerando grandes problemas de inconsistência. Mediante a todos estes problemas, torna os referidos procedimentos impraticáveis.

A seguir, é apresentada uma proposta de solução para os referidos problemas, utilizando o padrão de projeto *Abstract Factory*. Este padrão utiliza de forma intensa o princípio do Polimorfismo, permitindo que a aplicação seja totalmente adaptável ás mudanças, decorrentes durante o desenvolvimento, de

forma transparente.

```
1  <?
2      abstract class factoryAbstractBD {
3
4          protected function __construct() {
5
6          }
7
8          //Método de instanciacão (Abstrato)
9          abstract public function criaInstancia();
10     }
11 ?>
```

Figura 6.8 – Código fonte responsável pela criação da Classe *factoryAbstractBD*

Primeiramente, é criado a classe *factoryAbstractBD* conforme Fig.6.8. Esta classe é abstrata contendo o método de instanciação, o qual deverá ser sobrescrito em suas classes filhas, por isso o método *criaInstancia()* foi declarado como abstract.

Na Fig.6.9 é criado a classe "concreta" que herdará da classe *factoryAbstractBD* (Fig.6.8), e será responsável por criar objetos de conexão com o MySQL, o nome da classe é *factoryAbstractBDMysql* (Fig.6.9).

```
1  <?
2  class factoryAbstractBDMysql extends factoryAbstractBD {
3      public function __construct() {
4          parent::__construct();
5      }
6      public function criaInstancia() {
7          $db = new  MySQL ("user","senha");
8          //Foi omitido a implementação
9          //da classe MySQL por simplicidade
10         return $db;
11         //retorna um objeto de conexão MySQL
12     }
13 }
14 ?>
```

Figura 6.9 – Código fonte responsável pela criação da Classe *factoryAbstractBDMysql*

Na Fig.6.9, foi construída a primeira fábrica responsável por criar conexões com o MySQL. Então para alterar a senha e/ou usuário do banco de dados MySQL, é suficiente alterar somente a linha 7 da Fig.6.9 que corresponde ao usuário e senha do banco, e toda a aplicação será alterada com transparência e sem custo.

Na próxima etapa é criada a fábrica de objetos responsáveis para conexões com o banco de dados PostgreSQL (Fig.6.10). Tanto a classe *PostgreSQL* linha 7,

quanto a classe *MySQL* Fig.6.9 linha 7 devem herdar de uma mesma classe mãe, contendo métodos de manipulação de banco de dados. Estes métodos foram omitidos para maior simplicidade e serão implementados no item 6.8 deste cápitulo.

```
1  <?
2  class factoryAbstractBDPostgresql extends factoryAbstractBD {
3      public function __construct() {
4          parent::__construct();
5      }
6      public function criaInstancia() {
7          $db = new PostgreSQL ("user","senha");
8          //Foi omitido a implementação
9          //da calsse PostgreSQL por simplicidade
10         return $db;
11     }
12 }
13 ?>
```

Figura 6.10 – Código fonte responsável pela criação da Classe *factoryAbstractBDPostgresql*

Após a criação das fábricas, é demonstrada através do código fonte contido nas Fig.6.11 e Fig.6.12 a utilização do padrão. Para isso é criada uma classe que recebe como parâmetro um objeto fábrica, esta classe é responsável por executar qualquer consulta SQL e será chamada de *ExecutaFabrica*.

```
1  <?
2  class executaFabrica {
3      public function __construct() {
4
5      }
6      public function executa($fabrica,$query) {
7          $objetoBanco = $fabrica->criaInstancia();
8          //Não sabe qual fábrica a ser executada
9          $objetoBanco->executa($query);
10     }
11 }
12 ?>
```

Figura 6.11 – Código fonte responsável pela criação da Classe *executaFabrica*

Também é criada uma classe para teste (Fig.6.12) chamada de *teste*, que será responsável pela criação das duas Fábricas e passará, como parâmetro, um objeto dessas fábricas para a classe *executaFabrica* (Fig.6.11) e finalmente, esta classe executará diretivas de qualquer um dos banco de dados.

```
1  <?
2  class teste {
3      public function __construct() {
4      }
5      public function  main() {
6        $fabMySql = new factoryAbstractBDMyaql();
7        $fabPostgreSql = new factoryAbstractBDPostgresql();
8
9        $objeto = new ExecutaFabrica();
10
11       $query = ("SELECT * FROM tabela_exemplo");
12
13       $objeto->executa($fabMySql,$query);//Executando query em MySql
14       $objeto->executa($fabPostgreSql,$query);//Executando query em PostgreSql
15     }
16 }
17 ?>
```

Figura 6.12 – Código fonte responsável pela criação da classe teste

A classe *executaFabrica* (Fig.6.11) não tem o conhecimento com qual banco de dados está trabalhando, ou seja, não sabe qual o tipo concreto da fábrica passada (Fig.6.12 linhas 13 e 14). Portanto a aplicação tornou-se adaptável a mudanças de forma dinâmica e também reutilizável.

6.3 Estudo de caso do *Adapter*

No estudo de caso de utilização do padrão de projeto *Adapter* é implementada uma solução para que uma aplicação, que utiliza bibliotecas terceirizadas, mantenha-se sincronizada quando houver alterações essências nas funcionalidades daquelas bibliotecas.

Na solução apresentada a seguir, são utilizadas classes que trabalharão em uma camada intermediária entre a aplicação cliente e a biblioteca terceirizada, que serão responsáveis por manter a aplicação sincronizada. Esta biblioteca terceirizada é uma classe (Fig.6.13) utilitária de um framework especializado em métodos de validação simples.

```php
<?
/*
* Classe terceirizada de Validação
* Conjunto de métodos que validam determinados formatos de valores.
*/
class LibValidacao {
    //Validaçao de texto
    function isText($par){
        //Lógica de validação para texto
    }
    //Validação de endereço de e-mail
    function isEmail($par){
        //Lógica de validação para e-mail
    }
    //Validação de números inteiros
    function isInt($par){
         //Lógica de validação para números inteiros
    }
    //Validação para números no formato float
    function isFloat($par){
        //Lógica de validação para números no formato float
    }
    //Validação de data
    function isDate($par){
        //Lógica de validação para data
    }
    //Validação de hora
    function isTime($par){
        //Lógica de validação para hora
    }
}
?>
```

Figura 6.13 – Classe *LibValidacao* da biblioteca terceirizada

Supondo que na documentação da classe o método *isTime* (Fig.6.13) linha 28 seja depreciado no futuro e que em novas versões da classe *LibValidacao,* este método tenha sido substituído por um método com o nome *isHora*. Se apenas for substituída a nova classe lançada na aplicação, conseqüentemente iria ser gerado um erro, relativo a chamada de um método inexistente. Então, é neste momento em que o padrão de projeto *adapter* surge como solução.

Na implementação do padrão de projeto *adapter* é criada uma classe adaptadora (Fig.6.14) que herdará da classe utilitária *LibValidacao*, onde será sobrescritos os métodos da biblioteca.

```php
<?
/*
* Classe adaptadora da bibliotéca LibValidacao
*/
class ClasseAdapter_LibValidacao extends LibValidacao{
    public  function __construct() {
        parent::__construct();
    }
    //Validaçao de texto
    function isText($par){
        $this->isText($par);
    }
    //Validação de endereço de e-mail
    function isEmail($par){
        $this->isEmail($par);
    }
    //Validação de números inteiros
    function isInt($par){
         $this->isInt($par);
    }
    //Validação para números no formato float
    function isFloat($par){
        $this->isFloat($par);
    }
    //Validação de data
    function isDate($par){
        $this->isDate($par);
    }
    //Validação de hora
    function isTime($par){
        //READAPTANDO A CHAMADA PARA O NOVO MÉTODO
        $this->isHora($par);
    }
}
?>
```

Figura 6.14 – Classe adaptadora da biblioteca *LibValidacao*

É importante observar que na classe adaptadora (Fig.6.14) é realizado o sincronismo da aplicação com a biblioteca terceirizada, onde de forma transparente é substituído qualquer referencia da classe antiga *LibValidacao* por essa classe. Ou seja, na aplicação cliente, onde existir a utilização da classe *LibValidacao*, deve ser trocado pela *ClasseAdapter_LibValidacao* mantendo a coesão da aplicação. É importante observar que mesmo com a alteração de um nome de método na biblioteca terceirizada (Fig.6.14 linha 32) o sincronismo entre elas foi mantido.

A classe *Cliente*, descrita na Fig.6.15, representa a classe que utilizará a biblioteca através da classe Adapter criada Fig.6.14.

```php
<?
class Cliente {
    public  function __construct() {
        $hora = "07:00";

        //CÓDIGO ANTIGO
        //Esse código utiliza a antiga versão da LibValidacao
        $lib_antiga = new LibValidacao();
        if($lib_antiga->isTime($hora))
            return true;
        else
            return false;
        //-----------------------------------------------------

        //CÓDIGO NOVO
        //Esse código utiliza a versão adaptada da bibliotéca LibValidacao
        $lib_adaptada = new ClasseAdapter_LibValidacao();
        if($lib_adaptada->isTime($hora))
            return true;
        else
            return false;
        //-----------------------------------------------------
    }
}
?>
```

Figura 6.15 – Classe *cliente* utilizando a classe adaptada

6.4 Estudo de caso do *Composite*

No estudo de caso de aplicação do padrão de projeto *composite* é utilizada uma solução modificada e adaptada de Truett (2005), de modo que a aplicação trate objetos individuais e composições de objetos uniformemente.

Na solução apresentada a seguir, a classe *Livro* (Fig.6.17) é o objeto individual representando objetos folha na composição. A classe *diversosLivros* é um grupo composto por zero ou mais objetos Livro, representando o *Composite*. O componente é representado pela classe *livroNaPrateleira* (Fig.6.16), que declara a interface para objetos da composição, responsável por implementar o comportamento padrão comum a todas as classes.

```php
<?
abstract class livroNaPrateleira {
    abstract function getLivroInfo($livro);
    abstract function getLivroCount();
    abstract function setLivroCount($new_count);
    abstract function addLivro($livro);
    abstract function removeLivro($livro);
}
?>
```

Figura 6.16 – Classe *livroNaPrateleira*

```php
<?
class livro extends livroNaPrateleira {
    private $titulo;
    private $autor;

    public function __construct($titulo, $autor) {
        $this->titulo = $titulo;
        $this->autor = $autor;
    }

    public function getLivroInfo($livroParaPegar) {
        if (1 == $livroParaPegar)
            return $this->titulo." por ".$this->autor;
        else
            return false;
    }

    public function getLivroCount() {
        return 1;
    }

    public function setLivroCount($new_count) {
        return false;
    }

    public function addLivro($livro) {
        return false;
    }

    public function removeLivro($livro) {
        return false;
    }
}
?>
```

Figura 6.17 – Classe *livro*

```php
<?php
class diversosLivros extends livroNaPrateleira {
    private $livros = array(); private $livroCount;
    public function __construct() {
        $this->setLivroCount(0);
    }
    public function getLivroCount() {
        return $this->livroCount;
    }
    public function setLivroCount($new_count) {
        $this->livroCount = $newCount;
    }
    public function getLivroInfo($livroParaPegar) {
        if ($livroParaPegar <= $this->livroCount)
            return $this->livros[$livroParaPegar]->getLivroInfo(1);
        else
            return false;
    }
    public function addLivro($livro) {
        $this->setLivroCount($this->getLivroCount() + 1);
        $this->livros[$this->getLivroCount()] = $livro;
        return $this->getLivroCount();
    }
    public function removeLivro($livro) {
        $counter = 0;
        while (++$counter <= $this->getLivroCount()) {
            if ($livro->getLivroInfo(1)==$this->livros[$counter]->getLivroInfo(1)){
                for ($x = $counter; $x < $this->getLivroCount(); $x++) {
                    $this->livros[$x] = $this->livros[$x + 1];
                }
                $this->setLivroCount($this->getLivroCount() - 1);
            }
        }
        return $this->getLivroCount();
    }
}
?>
```

Figura 6.18 – Classe *diversosLivros*

É importante observar que ambas as classes *livro* Fig.6.17 e *diversosLivros* Fig.6.18 possuem os métodos *addLivro* e *removeLivro*, porém eles são funcionais somente sobre a classe *diversosLivros.* A classe *Livro* apenas irá retornar false quando aqueles métodos forem chamados.

O código descrito a seguir Fig.6.19 é mostrado uma seqüência de execuções dos métodos apresentados anteriormente, com o objetivo de realizar uma demonstração de teste do estudo de caso apresentado.

```php
<?php
    //CÓDIGO DE TESTE DO COMPOSITE

    $livro1 =  new livro("Livro 1", "Autor 1");
    $livro1->getLivroInfo(1); //Obtendo informações do livro 1

    $livro2 =  new livro("Livro 2", "Autor 2");
    $livro2->getLivroInfo(1); //Obtendo informações do livro 2

    $livros = new diversosLivros(); //Criando composição de vários livros

    //Adicionando o livro 1 a composição
    $livrosCount = $livros->addLivro($livro1);
    //Obtendo informações do livro 1 na composição
    $livros->getLivroInfo($livrosCount);

    //Adicionando o livro 2 a composição
    $livrosCount = $livros->addLivro($livro2);
    //Obtendo informações do livro 2 na composição
    $livros->getLivroInfo($livrosCount);

    //Removendo o livro 1 da composição
    $livrosCount = $livros->removeLivro($livro1);
    //Obtendo informações do livro 2
    $livros->getLivroCount(1);
?>
```

Figura 6.19 – Código de teste do estudo de caso do *Composite*

A partir do código contido na Fig.6.19 foi possível demonstrar na prática a essência do padrão de projeto *Composite*, na qual é possível através dele, tratar objetos primitivos linhas 4 e 7 da Fig.6.19 e composições de objetos linhas 13 e 18 Fig.6.19 de maneira a compartilhar uma mesma interface uniformemente.

6.5 Estudo de caso do *Decorator*

No estudo de caso de utilização do padrão de projeto *Decorator*, no desenvolvimento web, é implementado uma solução com o objetivo de manipular e modificar *strings* dentro de uma aplicação.

Na solução apresentada a seguir, o *decorator* é uma classe que permite adicionar mais capacidade para uma classe base já existente, usando alguns métodos intermédios, deixando intacta a classe original. Primeiramente, é criada a classe base (Fig.6.20), onde posteriormente terá suas funcionalidades estendidas por uma classe decoradora.

```php
<?php
//Classe base sendo criada (esta classe será decorada mais tarde)
class string{
    private $caminho;
    private $str;

    public function __construct($caminho,$str){
        $this->caminho = $caminho;
        $this->str = $str;
    }
    public function save(){
        $fp = fopen($this->caminho,'w');
        fwrite($fp,$str);
        flose($fp);
    }
    public function getCaminho(){
        return $this->caminho;
    }
    public function getString(){
        return $this->str;
    }
}
?>
```

Figura 6.20 – Código da classe base *string*

Posteriormente a criação da classe base, será criada uma classe a qual estendera a funcionalidade daquela classe sem alterar sua estrutura, e também sem a criação de uma sub-classe a partir dela. Para isso será utilizada a abordagem proposta pelo padrão de projeto *decarotor*.

Na Fig.6.21 é definida uma classe *decorator* chamada de *StringDecorator* que irá assumir a propriedade $str (Fig.6.20 linha 5), da classe base e associá-la com a sua própria propriedade, agindo como uma ponte para a ação do *decorator*.

```php
<?php
//Classe decorator sendo criada
class StringDecorator{
    protected $string;
    public $str;

    public function __construct(string $string){
        $this->string = $string;//Cópia do objeto original
        $this->resetString();
    }
    //Obtendo a sequencia do objeto `string`
    //Assim a propriedade original permanece a mesma
    public function resetString(){
        $this->str = $this->string->getString();
    }
    //Retorna a string
    public function mostraString(){
        return $this->str;
    }
}
?>
```

Figura 6.21 – Código da classe *StringDecorator*

A classe *StringDecorator* (Fig.6.21) recebe como parâmetro um objeto do tipo `*string*`, que então é atribuído a propriedade $string (Fig. 6.21 linha 8) da classe. O método *resetString()* utiliza o método *getString(),* pertencente a classe base *string*, para obter a propriedade $str. Desta forma é mostrado como construir uma classe que absorve as propriedades de uma classe base sem modificar a sua estrutura original. Até o momento a classe *StringDecorator* limita-se a tomar a propriedade $str da classe base. A seguir, a propriedade original $str ira ser decorada, por sub-classes concretas da classe *StringDecorator.*

Uma vez que a classe *StringDecorator* tenha sido definida, estendendo a funcionalidade da classe base *string*, serão criadas sub-classes a partir da classe *StringDecorator* para modificar a *string* de entrada, de acordo com a necessidade. Então neste estudo de caso, foi escolhido por simplicidade transformar a *string* de entrada para letras maiúsculas ou letras minúsculas, respectivamente através de sub-classes *StringMaiusculasDecorator* (Fig.6.22 linha 2) e *StringMinusculasDecorator* (Fig.6.22 linha 14).

```php
<?php
class StringMaiusculasDecorator extends StringDecorator{
    private $strDecorator;

    public function __construct(StringDecorator $strDecorator){
        $this->strDecorator = $strDecorator;
    }
    //Convertando a string para maiúsculas
    public function converteStringParaMaiusculas(){
        $this->strDecorator->str = strtoupper($this->strDecorator->str);
    }
}

class StringMinusculasDecorator extends StringDecorator{
    private $strDecorator;

    public function __construct(StringDecorator $strDecorator){
        $this->strDecorator=$strDecorator;
    }
    //Convertendo a string para minúsculas
    public function converteStringParaMinusculas(){
        $this->strDecorator->str = strtolower($this->strDecorator->str);
    }
}
?>
```

Figura 6.22 – Código das sub-classes StringMaiusculasDecorator e StringMinusculasDecorator

As duas classes acima listadas são sub-classes da *StringDecorator*, e aceitam este objeto como único parâmetro de entrada, onde essas modificam a propriedade $str mantendo a estrutura da classe base claramente intocável. Assim fazendo uso da abordagem proposta pelo padrão de projeto *Decorator*.

6.6 Estudo de caso do *Observer*

No estudo de caso de aplicação do padrão de projeto *Observer*, a aplicação em questão é responsável por notificar todos os participantes de um fórum, através de e-mail e rss, quando algum novo post é adicionado.

Na solução apresentada a seguir, primeiramente são definidas duas classes de interface, a classe *subject* (Fig.6.23) na qual será herdada pela classe *post* e lhe permitindo tornar-se um objeto de observação e a classe *observer* (Fig.6.24) que será herdada pelas classes usadas para observar um objeto *post*.

```php
<?php
//Classe geradora de objetos a serem observados(Post)
class subject {
    //Objetos a serem notificados pelo post
    var $observadores;
    //Armazena o estado do post
    var $estado;

    public function __construct() {
        $this->observadores=array();
    }

    //Método para notificar observadores de Post
    function notificarObservadores() {
        $observadores = sizeof($this->observadores);
        for ($i=0;$i<$observadores;$i++) {
            $this->observadores[$i]->update();
        }
    }

    //Adiciona observadores a um post
    function addObservador (observer $observer) {
        $this->observadores[] = $observer;
    }

    //Retorna o estado atual de um post
    function getEstado () {
        return $this->estado;
    }

    //Seta o estado atual de um post
    function setEstado ($estado) {
        $this->estado=$estado;
    }
}
?>
```

Figura 6.23 – Classe *subject* sendo criada

```php
<?php
//Classe Observer
class observer {
    //Instância do objeto a ser observado(Post)
    var $subject;

    public function __constructr (subject $subject) {
        $this->subject = $subject;
        // Registra este observador para poder notifica-lô
        $subject->addObservador($this);
    }

    function update() {
        //Método deve ser implementado pelas sub-classes
        //Este método executara uma ação específica de cada observador
    }
}
?>
```

Figura 6.24 – Classe *observer* sendo criada

Posteriormente as implementações das duas classes bases, são criadas as classes *post* (Fig.6.25) na qual representa um *post* criado na aplicação e também as classes *email* e *rss,* que serão observadoras de *post* e responsáveis por enviar e-mails e *rss* de notificações aos usuários do fórum.

```php
<?php
//Classe post
class post extends subject {
    public function __construct(){
        subject::subject();
    }

    //Adicionando um post
    function addPost(){
        //Não esta implementado o método que adiciona o post no BD

        $this->setEstado("Post adicionado");

        //Notifica todos os seus observadores
        $this->notificarObservadores();
    }
}
?>
```

Figura 6.25 – Classe *post* sendo criada

O método *addPost()* (Fig.6.25 linha 9) chama o método *notificarObservadores()* da sua classe base *subject* na Fig.6.23, significando que quando um post é adicionado todos os seus observadores serão notificados e reagirão adequadamente. É importante observar que um post não tem conhecimento de quem o observa, pois isso é tratado na classe *observer* que é estendida pelos

observadores.

```php
<?php
//Classe email
class email extends Observer {

    public function __construct(subject $subject){
        observer::observer($subject);
    }

    //Método de reação ao adicionar um novo post
    //Este método é chamado por notificarObservadores()
    function update() {
        if($this->subject->getState() == "Post adicionado" ){
            $this->sendMail();
        }
    }

    //Envia um e-mail para os usuários do fórum
    function sendMail() {
        //Método não foi implementado por simplicidade
    }
}
?>
```

Figura 6.26– Classe *email* sendo criada

Na Fig.6.26 é criada a classe e-mail na qual é responsável por notificar os usuários do fórum quando um novo post é adicionado. Na linha 11 da Fig.6.26 o método *update()* é o método de reação a adição de um novo *post*, este método é chamado transparentemente pelo método *noticarObservadores()* na Fig.6.25 linha 15.

```php
<?php
//Classe rss
class rss extends Observer {

    public function __construct (subject $subject) {
        observer::observer($subject);
    }

    //Método de reação ao adicionar um novo post
    //Este método é chamado por notificarObservadores()
    function update () {
        if ($this->subject->getState() == "Post adicionado" ) {
            $this->updateRss();
        }
    }

    //Atualiza o RSS do fórum
    function updateRss() {
        //Mátodo não foi implementado por simplicidade
    }
}
?>
```

Figura 6.27 – Classe *rss* sendo criada

Na Fig.6.27 é criada a classe rss na qual é responsável por atualizar a sessão de RSS do fórum quando um novo post é adicionado. Na linha 11 da Fig.6.27 o método *update()* é o método de reação a adição de um novo post, este método é chamado transparentemente pelo método *noticarObservadores()* na Fig.6.25 linha 15.

```php
<?php
//CÓDIGO DE TESTE DO OBSERVER

    //Cria um novo post (Objeto a ser observado)
    $post = new post;

    //Cria os observadores
    $email = new email($post);
    $rss = new rss($post);

    //Chamando o método addPost()
    //Aciona os observadores para responder
    //através do método notificarObservadores()
    $post->addPost();
?>
```

Figura 6.28 – Código de teste do observer

Na Fig.6.28 as instâncias $mail e $rss, linhas 8 e 9 respectivamente, respondem automaticamente ao evento de adicionar um novo *post*. Quando elas foram instanciadas, registraram-se como observadoras de *Post*, basicamente

dizendo “desejo saber todas as suas alterações”. Quando o método *addPost()*, na linha 14 da Fig.6.28 foi chamado, *Post* notificou email e rss de que tinha ocorrido uma mudança de estado, deixando-os decidir que ação tomar. Assim sendo, foi demonstrada a abordagem proposta pelo padrão de projeto *observer*, no desenvolvimento de aplicações Web.

6.7 Estudo de caso do *Strategy*

No estudo de caso de utilização do padrão de projeto *Strategy* é implementado uma solução, com o objetivo de otimizar, em uma aplicação Web, a forma como os campos enviados por um formulário são validados, evitando uma interminável reprodução de declarações de if / else que são realizadas quando não se utiliza esse padrão, neste tipo de validação.

No estudo de caso apresentado a seguir é construído um conjunto de classes acessível através de uma interface única a superclasse, chamada de *validação* (Fig.6.29), as sub-classes são implementadas para cada tipo de validação que se tem necessidade. As subclasses encapsulam os algoritmos específicos para cada validação, e para adicionar mais tipos de validação basta adicionar mais sub-classes para a necessidade. Neste estudo de caso serão implementados somente validação de nomes de usuário, senha e e-mail.

Primeiramente é construída a superclasse *validacao* (Fig.6.29):

```php
<?php
//Classe validacao
class validacao {
    //Array de possíveis erros
    var $erro;

    public function __construct(){
        $this->erro = array();
        $this->validar();
    }

    function validar() {
        //Método é implementado pelas sub-classes
    }

    //Seta a mensagem de erro
    function setErro($msg) {
        $this->erro[] = $msg;
    }

    //Captura o erro
    function getErro() {
        return $this->erro;
    }

    //Verifica se existe mensagem de erro
    function isValido () {
        if (sizeof($this->erro) > 0)
            return false;
        else
            return true;
    }
}
?>
```

Figura 6.29 – Superclasse *validacao*

A classe pai *validação* (Fig.6.29), define uma interface pela qual qualquer código cliente tem acesso à forma de validação padrão. Posteriormente são implementadas as sub-classes que serão acessíveis através da interface criada na Fig.6.29.

```php
<?php
//Classe de validação de nome usuário
class validaUsuario extends validacao {
    var $usuario;

    public function __construct($usuario){
        $this->usuario = $usuario;
        validacao::validacao();
    }

    //Método de validação de nome de usuário
    function validar() {
        if (!preg_match('/^[a-zA-Z0-9_]+$/',$this->usuario )){
            $this->setErro("Nome do usuário possui caracteres inválidos");
        }
        if (strlen($this->usuario) < 4 ){
            $this->setErro("Nome do usuário pequeno");
        }
    }
}
?>
```

Figura 6.30 – Sub-classe de validação de nome de usuário *validaUsuario*

```php
<?php
//Classe de validação de password
class validaPassword extends validacao {
    var $password;
    var $confirmaPass;

    public function __construct($pass,$conf){
        $this->password = $password;
        $this->confirmaPass=$confirmaPass;
        validacao::validacao();
    }

    //Método de validação de password
    function validar() {
        if ($this->password != $this->confirmaPass){
            $this->setErro("Password não confere");
        }
        if (!preg_match('/^[a-zA-Z0-9_]+$/',$this->password )){
            $this->setErro("Password contem caracteres inválidos");
        }
        if (strlen($this->password) < 8 ) {
            $this->setErro("Password muito pequeno");
        }
    }
}
?>
```

Figura 6.31 – Sub-classe de validação de password *validaPassword*

```php
<?php
//Classe de validação de e-mail
class validaEmail extends validacao {
    var $email;

    public function __construct($email){
        $this->email = $email;
         validacao::validacao();
    }

    //Método de validação de e-mail
    function validar() {
        $er = "/^([a-zA-Z0-9])+([.a-zA-Z0-9_-])*@([a-zA-Z0-9_-])+(.[a-zA-Z0-9_-]+)+/";
        if(!preg_match($er,$this->email)){
            $this->setErro("Endereço de e-mail inválido");
        }
        if (strlen($this->email) > 80){
            $this->setErro("Endereço de e-mail longo");
        }
    }
}
?>
```

Figura 6.32 – Sub-classe de validação de e-mail *validaEmail*

```php
<?php
//CÓDIGO DE TESTE DO STRATEGY
if ($_POST["form"]){ //Formuário submetido

    //Sub-classes de validação
    $arr[] = new validaUsuario($_POST['usuario']);
    $arr[] = new validaPassword($_POST['password'],$_POST['confirmaPass']);
    $arr[] = new validaEmail($_POST['email']);

    //Concatenando mensagens de erro, caso exista
    $erros = "";
    foreach($arr as $validando) {
        if (!$validando->isValido()) {
            if($validando->getErro()) {
                $erros.= $validando->getErro()."<br>";
            }
        }
    }
    if ($erros) {
        print ("Erros encontrados:".$erros);
    } else {
        print ("Formulário válido");
    }
}
?>
```

Figura 6.33 – Código de teste do padrão de projeto *Strategy*

Na Fig.6.33 as classes linhas 6, 7 e 8 fazem uso de uma interface única provida pela classe *validação* (Fig.6.29), onde esta classe possui os métodos básicos, que são padrão para todas as suas sub-classes, com exceção do método *validar()* (linha 12 da Fig.6.29), que é implementado em cada sub-classe de acordo a necessidade de validação dela. É importante observar que caso se queira adicionar

mais campos no formulário, basta adicionar mais sub-classes para tratar estes novos campos e implementar seus métodos de validação *validar()*, pois o restante já esta implementado na superclasse *validacao()* (Fig.6.29), que encapsula as funções básicas do processo de validação conforme a abordagem do padrão de projeto *Strategy.* Através deste estudo de caso foi possível demonstrar a utilização do padrão de projeto *Strategy*, no desenvolvimento de aplicações Web.

6.8 Estudo de caso do *Template Method*

Na aplicação do padrão de projeto *Template Method* é realizado um estudo de caso complementar ao do item 6.2 deste cápitulo, que aborda o padrão Abstract Factory, onde, naquela implementação as classes responsáveis por gerar objetos de banco de dados MySQL e PostgreSQL foram omitidas, (Fig.6.9 linha 7 e Fig.6.10 linha 7) respectivamente. Neste estudo de caso é realizada uma sugestão de implementação para estas duas classes, com a possibilidade de se adicionar mais bancos de dados, a uma aplicação, caso seja necessário.

No estudo de caso apresentado a seguir, primeiramente é construída a classe abstrata responsável por definir o *template* de todas as suas sub-classes, esta classe será chamada de *bancoDeDados* (Fig.6.34), onde para qualquer banco de dados que se desejar utilizar na aplicação, a classe responsável por implementar este banco, terá de ser, obrigatoriamente, estendida da classe *bancoDeDados* e implementar todos os métodos por ela definidos.

```php
<?
//Classe abstrata, template para as sub-classes.
//Métodos que obrigatóriamente deverão ser
//implementados pelas subclasses.
abstract class bancoDeDados {

	private $conexao;
	private $resultado;

	protected function __construct();

	//Executar cláusula SQL
	abstract function executa($query);

	//Obter número de resultados de uma consulta SQL
	abstract function numeroDeRegistros();

	//Obter informações da consulta mais recente
	abstract function info();

	//Obtém id gerado pela operação INSERT anterior
	abstract function ultimoIdGerado();
}
?>
```

Figura 6.34 – Classe abstrata *bancoDeDados*

Na Fig.6.34 é implementada a classe *bancoDeDados* que é uma classe de template, onde qualquer classe responsável por implementar banco de dados da aplicação terá de ser estendida dessa classe, e terá de implementar todos os métodos descritos por ela.

```php
<?
class MySQL extends bancoDeDados {
     private $conexao;
     private $resultado;

     function __construct($usuario, $senha){
        $conexao = mysql_connect('localhost', $usuario, $senha);
        if (!$conexao)
            exit("Não foi possível conectar ao banco de dados");
        $this->conexao = $conexao;
        return $this;
     }
     //Executar cláusula MySQl
     function executa($query){
        $resultado = mysql_query($query);
        if (!$resultado)
            exit("Query MySQL inválida");
        $this->resultado = $resultado;
     }
     //Obtém número de resultados de uma consulta MySQL
     function numeroDeRegistros(){
        if($this->resultado)
            return mysql_num_rows($this->resultado);
        else
            return false;
     }
     //Obtém informações da consulta mais recente
     function info(){
        return  mysql_info($this->conexao);
     }
     //Obtém id gerado pela operação INSERT anterior
     function ultimoIdGerado(){
        return  mysql_insert_id($this->conexao);
     }
}
?>
```

Figura 6.35 – Classe concreta *MySQL*

Na Fig.6.35 a classe *MySQL* é estendida da classe de *template* padrão *bancoDeDados* (Fig.6.34), aquela classe implementa todos os métodos abstratamente definidos pela classe padrão, porém de acordo com as diretivas de conexão para banco de dados MySQL.

```php
<?
class PostgreSQL extends bancoDeDados {
     private $conexao;
     private $resultado;

     function __construct($usuario, $senha){
        $conexao = pg_connect('localhost', $usuario, $senha);
        if (!$conexao)
            exit("Não foi possível conectar ao banco de dados");
        $this->conexao = $conexao;
        return $this;
     }
     //Executa cláusula PostgreSQL
     function executa($query){
        $resultado = pg_query($this->conexao,$query);
        if (!$resultado)
            exit("Query PostgreSQL inválida");
        $this->resultado = $resultado;
     }
     //Obtém número de resultados de uma consulta PostgreSQL
     function numeroDeRegistros(){
        if($this->resultado)
            return pg_num_rows($this->resultado);
        else
            return false;
     }
     //Obtém informações da consulta mais recente
     function info(){
        return  pg_result_status($this->conexao);
     }
     //Obtém id gerado pela operação INSERT anterior
     function ultimoIdGerado(){
        return  pg_last_oid($this->conexao);
     }
}
```

Figura 6.36 – Classe concreta *PostgreSQL*

Na Fig.6.36 a classe *PostgreSQL* também é estendida da classe de *template* padrão *bancoDeDados* (Fig.6.34), e implementa todos os métodos abstratamente definidos pela classe padrão, porém de acordo com as diretivas de conexão para banco de dados *PostgreSQL*. É importante observar que para se adicionar um novo banco de dados na aplicação é necessário que a classe que implementará este novo banco de dados seja estendida da classe de *tamplate*, desta forma a aplicação se manterá robusta e consistente, pois todas as chamadas de diretivas de conexão para banco de dados possuem um mesmo formato pré-definido pela classe *template* padrão, conforme a abordagem proposta pelo padrão de projeto *template method*.

7 Considerações finais

O presente trabalho apresentou estudos de caso de aplicação de alguns padrões de projeto no desenvolvimento de aplicações em ambiente Web. Também mostrou uma análise do atual estágio de utilização desses padrões por parte de algumas empresas deste ramo na cidade de Pelotas.

Os padrões de projeto abordados por este trabalho, foram selecionados através de uma pesquisa de campo, na qual foram apontados oito padrões como sendo os mais utilizados nas empresas investigadas. Os padrões selecionados foram: *Abstract Factory, Factory Method, Adapter, Composite, Decorator, Observer, Strategy e Template Method.*

Para cada padrão de projeto do grupo selecionado foi realizado um estudo de caso de aplicação deste padrão. O *Abstract Factory* foi aplicado para atender a necessidade de conexão com mais de um banco de dados em uma aplicação Web, o *Factory Method* foi utilizado para dinamizar a maneira como os *requires/includes* das classes que compõem uma aplicação são realizados, o *Adapter* foi empregado para manter uma aplicação sincronizada a bibliotecas terceirizadas, o *Composite* foi utilizado de modo que uma aplicação trate objetos individuais e composições de objetos uniformemente, o *Decorator* foi aplicado com o objetivo de manipular e modificar *strings* dentro de uma aplicação Web, o *Observer* foi utilizado em uma aplicação Web, onde esta é responsável por notificar todos os participantes de um fórum, através de e-mail e rss, quando algum novo post é adicionado, o *Strategy* foi aplicado para otimizar a forma como os campos enviados por um formulário Web são validados e o *Template Method* foi utilizado em um estudo de caso complementar ao do *Abstract Factory*, com o objetivo de estruturar e padronizar as classes responsáveis por implementar diretivas de conexão com todos os banco de dados de uma aplicação.

Como foi observado, ao longo do desenvolvimento deste trabalho, a utilização de padrões de projeto no desenvolvimento, especificamente, em ambiente Web,

vem maximizar os ganhos relativos a esforços, custos e qualidade do produto final e em contra partida minimizar o tempo de desenvolvimento. Fazendo com que se tenham produtos de software que se adeqüei melhor as metas de qualidade, e que sejam entregues dentro dos prazos pré-definidos. Além de permitir, caso necessário, um melhor apoio a manutenções futuras deste software.

O trabalho, fundamentalmente, na parte da pesquisa de campo, propiciou uma integração maior entre universidade e empresa, visto que foram realizados vários contatos e discussões com líderes de projeto de algumas empresas de desenvolvimento Web da cidade de Pelotas. Esta experiência se tornou muito importante, pois a partir dela, foi possível conhecer de perto o funcionamento de algumas empresas de desenvolvimento de software e conseqüentemente uma maior aproximação junto ao mercado de trabalho.

Porém, foram encontradas algumas dificuldades durante o desenvolvimento do trabalho, sem duvida a maior delas se deu com a dificuldade de se coletar os dados relativos à pesquisa de campo. Pois, os responsáveis por fornecer estes dados geralmente eram líderes de projeto e se apresentavam constantemente atarefados e com pouca disponibilidade de tempo.

Uma sugestão para trabalhos futuros seria analisar a utilização de padrões de projeto em outros ambientes diferentemente do Web, e também realizar a aplicação destes padrões utilizando outras linguagens de programação como Java.

8 Referências

ALEXANDER, C. et. Al. **A Pattern Language**, Oxford University Press. New York, 1977.

ALUR, D., CUPRI, J., MALKS, D. **As melhores práticas e estratégias de design.** Rio de Janeiro: Campus, 2002.

APPLETON, B**. Patterns and Software: Essential Concepts and Terminology**. Disponível em: <http://www.enteract.com/~bradappdocpatterns-intro.html>. Acesso: 15 out 2008

ARRINGTON, C. T. **Enterprise Java with UML**. New York: John Wiley & Sons, Inc., 2001.

BECK, K. **Extreme Programming Explained – Embrace Change**. Addison Wesley, 1999.

BECK, K., CUNNINGHAM, W. **Using Pattern Languages for Object-Oriented Programs**. Technical Report nº CR-87-43, 1987. Disponível em <http://c2.com/doc/oopsla87.html>. Acesso em: 05 Abr 2008

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M. **Pattern-Oriented Software Architecture: A System of Patterns**. Wiley, 1996.

BUSCHMANN, F. et al. **A System of Patterns**. Wiley, 1996.

CASANOVA a, Paulo T. Gomes; DINIZ, Eliane da S. Alcoforado. **Utilizando Padrões de Projeto no Desenvolvimento de Sistemas em Ambiente Web. Estudo de caso: Factory Method e Abtract Factory**. Jornada de Produção Cientifica e Tecnológica da Região Sul. Pelotas, 2008.

CASANOVA b, Paulo T. Gomes; DINIZ, Eliane da S. Alcoforado. **Utilizando Padrões de Projeto no Desenvolvimento de Sistemas em Ambiente Web**. Congresso de Iniciação Científica - CIC . Pelotas, 2008.

CAVALCANTI, A. **Qualidade de Software, teoria e prática.** Prentice Hall. 2001

COAD, P.; YOURDON, E. **Object-Oriented Analysis**, 2ª edição, Yourdon Press, 1991.

COLEMAN, D. et al. **Object Oriented Development - the Fusion Method**. Prentice Hall, 1994.

COOK, S.; DANIEL, J. **Designing Object Systems**. Prentice Hall, 1994.

DEITEL, H.; Deitel, P. J. **Java, Como Programar**. Porto Alegre: Bookman, 2001.

ERIKSSON, H.; PENKER, M. **UML Toolkit**. New York, Wiley Computer Publishing, 1998.

FOWLER, M. **Analysis Patterns: Reusable Object Models**. Addison-Wesley Professional, 1996. 384p.

GAMMA, E., HELM, R., JOHNSON, R. e VLISSIDES, J. **Design Pattenrs – Elements of Reusable Object-Oriented Software**. Addison Wesley, 1995.

GAMMA, Erich et al. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2000.

LOZANO, R. **Patterns e Anti-Patterns para o desenvolvimento em PHP**, Disponível em: <http://www.lozano.eti.br/palestras/patters-php.pdf>. Acesso: 28 set 2008

MESTKER, S. J. **Wzorce projektowe**. Helion, 2005.

MASIERO, P.C.; GERMANO, F.S; MALDONADO, J.C. **Object and System Life Cycles Revisited: Their Use in Object-Oriented Analysis and Design Methods**. Proceedings of the 3rd CaiSE/IFIP8.1 (International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design), Pisa, Italy, pp. O1-O12, Junho 1998.

RIEHLE, D.; ZULLIGHOVEN, H. **Understanding and Using Patterns in Software Developmen, in Theory and Practice of Object System**, Disponível em: <http://www.ubs.com/webclub/ubilab/staff/e_riehle.htm>. Acesso em: 09 set 2008

SUN MICROSYSTEM. **Core J2EE Patterns - Data Access Object**. Disponível em: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>. Acesso em: 01 Abr 2008.

SUN MICROSYSTEM. **Java blueprints guidelines, patterns, and code for end-to-end Java applications**.Disponível em: <http://java.sun.com/blueprints/*patterns*/index.html>. Acesso em: 30 mar 2008.

TURINE, A. S. **Fundamentos, Conceitos e Aplicações do Paradigma de Orientação a Objetos**. Apresentação didática, agosto de 199. Disponível em: <http://nt-labes.icmsc.sc.usp.br/cursos/sce220/02_98/aulas.html>. Acesso em: 10 out 2008.

TRUETT, H. **PHP Design Patterns Reference and Examples**, Disponível em: <http://www.fluffycat.com/PHP-Design-Patterns>. Acesso: 10 out 2008

APÊNDICE A - Questionário utilizado para pesquisa de campo

UNIVERSIDADE FEDERAL DE PELOTAS
Instituto de Física e Matemática
Departamento de Informática
Curso de Ciência da Computação

Questionário para pesquisa de campo

1. Dados do entrevistado

Nome:	
Cargo:	

2. Caraterização da empresa atuante

Nome:	
Tempo ne mercado:	
Nro. de funcionários:	
Nro. de clientes:	

3. Tecnologias utilizadas para desenvolvimento

Linguagem (s) de programação utilizada (s):	
Paradigma de programação:	Utiliza orientação a objetos? ( ) Sim ( ) Não
Framework (s) de desenvolvimento:	Utiliza framework (s) ? ( ) Sim ( ) Não Se sim, qual (s) ? (citar):

Observação

UNIVERSIDADE FEDERAL DE PELOTAS
Instituto de Física e Matemática
Departamento de Informática
Curso de Ciência da Computação

Questionário para pesquisa de campo

Assinalar as alternativas conforme a legenda abaixo:

A) Padrão é utilizado;
B) Padrão é utilizado sem conhecimento prévio;
C) Padrão não é utilizado.

4.Padrões de projeto utilizados (*Design Patterns*)

Abstract Factory # Cria uma família de objetos relacionados ou dependentes sem a necessidade de especificar a classe concreta destes objetos.	( ) A ( ) B ( ) C
Factory Method # Define uma interface para criação de objetos, mas deixa as subclasses decidirem qual classe ser instanciada, permitindo a uma classe postergar a instanciação ás subclasses.	( ) A ( ) B ( ) C
Adapter # Converte a interface de uma classe em outra esperada pelo cliente, permitindo que certas classes trabalhem em conjunto, pois de outra forma não seria possível por causa de suas interfaces incompativeis.	( ) A ( ) B ( ) C
Composite # Compõem objetos em estrutura de árvore para representar hierarquias do tipo parte-todo, permitindo que o cliente trate objetos individuais e composção de objetos de maneira uniforme.	( ) A ( ) B ( ) C
Decorator # Atribui responsabilidades adicionais a um objeto dinamicamente fornecendo uma alternativa flexível a subclasses para a extensão da funcionalidade.	( ) A ( ) B ( ) C
Observer # Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependetes são automaticamente notificados e atualizados.	( ) A ( ) B ( ) C
Strategy # Define uma família de algoritmos e encapsula cada um deles, tornando-os intercambiáveis, permitindo que o algoritimo varie independentemente dos clientes que os utilizam.	( ) A ( ) B ( ) C
Template Method # Define o esqueleto de um algoritimo, permitindo que as subclasses customizem o restante do algoritimo.	( ) A ( ) B ( ) C

Outros padrões de projeto utilizados (citar)	É utilizado?
	( ) Sim ( ) Não

APÊNDICE B - Autorizações de publicação dos dados da pesquisa

Assinaturas dos responsáveis de cada empresa, participante da pesquisa, autorizando a publicação dos dados coletados.

Conrad Caine

Checkplant

ADSWEB

Coinpel

REWEB

Centro de Informática

APÊNDICE C – Descrição das empresas participantes da pesquisa

- Checkplant Sistemas de Rastreabilidade - A CheckPlant é uma empresa focada em desenvolvimento de soluções informatizadas de rastreabilidade para produtos de origem animal e vegetal. Está no mercado desde 2001. Nasceu da inquietude de pesquisadores da área agrícola (Universidade Federal de Pelotas – UFPel) em associação com profissionais da área de Tecnologia da Informação, na busca de soluções inovadoras no manejo de fazendas frutíferas: uva, pêssego, maçã, manga, morango e citros. Atualmente sua equipe é formada por 15 funcionários e possui 20 clientes sendo a maioria localizados em Petrolina – PE no Vale do São Francisco. O responsável pelo fornecimento dos dados de pesquisa foi o diretor de tecnologia André Guerreiro Cantarelli.

- Conrad Caine Media Applications - A Conrad Caine é uma empresa focada principalmente no desenvolvimento de aplicações web. Esta no mercado de software a mais de 10 anos, tem sua sede localizada na cidade de München na Alemanha e possui um grande escritório de desenvolvimento na cidade de Pelotas. Atualmente sua equipe de desenvolvimento é formada por 50 desenvolvedores. Em seu rol de clientes pode-se destacar grandes marcas como Siemens, Microsoft entre outras o que configura uma empresa que desenvolve software de elevada qualidade. Esta empresa é de maior porte e estrutura participante da pesquisa. O responsável pelo fornecimento dos dados de pesquisa foi líder de projeto Henrique da Silva Vilela.

- REWEB - Essa empresa iniciou suas atividades em 2000, tendo como definição de negócio o desafio diário de redesenhar a web. Mais do que simplesmente inovar, a idéia é aliar design, sistemas, inteligência e tecnologia visando maximizar resultados para as empresas clientes usufruindo da evolução constante que esse mercado oferece. A equipe Reweb é formada por 15 funcionários incluindo profissionais de marketing, análise de sistemas, design, motion design, entre outros, visando oferecer muito mais do que uma ferramenta de comunicação

virtual, e sim uma estratégia de comunicação integrada uma solução completa e em permanente *upgrade*. Possui atualmente 45 clientes destacando um de seus maiores a rede de farmácias AGAFARMA. A responsável pelo fornecimento dos dados de pesquisa foi a gerente de TI Tagline Teichel.

- Companhia de Informática de Pelotas – Coinpel - Essa é uma empresa pública criada através da Lei Municipal nº 3.229 de 11/10/89, regulamentada pelo Decreto nº 2.596 de 10/11/89, está localizada na rua Félix da Cunha, 610, centro de Pelotas. Esta empresa tem por objetivo estudar e viabilizar tecnologias de informação e comunicação na área da administração pública direta e indireta, atuando na gestão dos processos e recursos destas tecnologias, compreendendo sistemas operacionais, aplicativos e equipamentos, proporcionando serviços de consultoria, processamento, tratamento e transmissão de informações, bem como o desempenho de atividades correlatas, para o Município de Pelotas. Atualmente sua equipe é formada por 24 funcionários e possui 5 clientes. O responsável pelo fornecimento dos dados de pesquisa foi o diretor técnico Eduardo Carpena.

- Soluções de Tecnologia da Informação – ADSWeb - Essa é uma empresa que surgiu da necessidade de desenvolvimento de sistemas na área da saúde em Hospitais de Ensino principalmente da UFPEL. Sua missão é atender a demanda de informações e tecnologia em diversas áreas. Apresenta como princípio vender o que utiliza e produzir sistemas que atendam às necessidades específicas de cada cliente. Atualmente sua equipe é formada por 11 funcionários sendo 1 analista, 4 programadores, 1 webmaster, 1 administrador de redes, 3 da área de suporte e 1 administrativo. Possui 2 clientes externos e presta serviço de desenvolvimento de sistemas Web para UFPEL. Os responsáveis pelo fornecimento dos dados de pesquisa foi a gerente Beatriz da Silva Ribeiro e o coordenador de desenvolvimento Marcelo Santos de Souza.

- Centro de Informática – É um órgão vinculado diretamente ao Gabinete do Reitor e tem por finalidade participar da definição da política de informática da Universidade Federal de Pelotas (UFPel), bem como, coordenar, supervisionar, dirigir e avaliar a execução dos projetos e atividades necessárias à implementação das diretrizes da área de informática da Instituição. Apresenta como objetivos: fornecer soluções, baseadas em tecnologia da informação, para o desenvolvimento dos processos de gestão da Universidade; coordenar e executar serviços de informática para as áreas acadêmica e administrativa da Instituição; gerenciar a rede corporativa da UFPel; administrar os serviços de Internet e Intranet da Instituição; elaborar, executar e apoiar programas institucionais de treinamento em informática; Apoiar a área acadêmica quanto ao uso da informática no processo ensino/aprendizagem e Integrar o Comitê de Informática da UFPel. O responsável pelo fornecimento dos dados de pesquisa foi o diretor João Ladislau Lopes e o desenvolvedor David Valente.