UNIVERSIDADE FUMEC
FACULDADE DE CIÊNCIAS EMPRESARIAIS - FACE

MATHEUS HIGINO DE OLIVEIRA CASSIMIRO

PADRÕES ARQUITETURAIS E SEUS BENEFÍCIOS NO PROCESSO DE MANUTENÇÃO DE SOFTWARE

BELO HORIZONTE
2010

MATHEUS HIGINO DE OLIVEIRA CASSIMIRO

PADRÕES ARQUITETURAIS E SEUS BENEFÍCIOS NO PROCESSO DE MANUTENÇÃO DO SOFTWARE

Monografia realizada na Universidade FUMEC, no curso de Ciência da Computação, apresentado à disciplina Trabalho de Conclusão de Curso.

Orientadores:
Professor Ricardo Terra
Professor Osvaldo Manoel Corrêa

BELO HORIZONTE
2010

MATHEUS HIGINO DE OLIVEIRA CASSIMIRO

PADRÕES ARQUITETURAIS E SEUS BENEFÍCIOS NO PROCESSO DE MANUTENÇÃO DO SOFTWARE

Monografia de conclusão de curso submetida à Universidade FUMEC como requisito parcial para obtenção do título de Bacharel em Ciência da Computação e aprovada pela seguinte banca examinadora:

_____
Professor Ricardo Terra (Orientador)
Universidade FUMEC

_____
Professor Osvaldo Manoel Corrêa (Professor TCC)
Universidade FUMEC

Data da aprovação:

BELO HORIZONTE
2010

“Everything should be made as simple as possible, but no simpler.” (Albert Einstein)

ABSTRACT

With the evolution of computing and its resources, corporative software systems are becoming each time more complex not only on its requirements, but also on their development and maintance. Such fact provokes a challenge to use only the necessary resources to guarantee the scalability, maintainability and portability of these systems.

This work searches on raising the importance of applying this resources to guarantee the excellency and quality desired for a corporative system, not only on its inicial developing process, but also at the maintance and alterations moment.

Such resources are the application of architectural and project patterns that assure a functional system drawing and also preserve its benefits on the maintance process.

KEYWORDS: Design patterns; Architectural patterns; Software maintance.

RESUMO

Com a evolução da computação e seus recursos, os sistemas de software corporativos vêm se tornando cada vez mais complexos não apenas nos seus requisitos, mas também em sua elaboração e manutenção. Tal fato provoca um desafio de utilizar os recursos necessários para garantir a escalabilidade, manutenabilidade e portabilidade desses sistemas.

Este trabalho busca levantar a importância de se aplicar esses recursos para garantir a excelência e qualidade desejada para o sistema corporativo, não apenas no seu processo de desenvolvimento inicial, mas também no momento de realizar-se alterações e manutenções.

Tais recursos são a aplicação dos padrões de projeto e arquiteturais para garantir um bom desenho do sistema e também preservar seus benefícios no processo de manutenção.

PALAVRAS CHAVES: Padrões de projeto; Padrões arquiteturais; manutenção de software.

LISTA DE FIGURAS

FIGURA 1: Padrão *Abstract Factory* 16

FIGURA 2: Padrão *Adapter*..... 20

FIGURA 3: Padrão strategy 23

FIGURA 4: Padrão Em Quatro Camadas..... 29

FIGURA 5: Padrão Em Três Camadas..... 30

FIGURA 6: Padrão MVC 33

FIGURA 7: Arquitetura Do Sistema Tmturismo..... 37

FIGURA 8: Padrão *Strategy* Na Camada De Negócio Do Sistema Mtturismo..... 39

LISTA DE TABELAS

TABELA 1: Classificação Dos Padrões De Projeto 15

SUMÁRIO

Introdução **10**

Capítulo 1 – Padrões De Projeto **13**

1.1 – O Que É Um Padrão De Projeto **13**

1.2 – Padrões De Criação **15**

1.3 – Padrões Estruturais **19**

1.4 – Padrões Comportamentais **22**

1.5 – Limitações Dos Padrões De Projeto **25**

Capítulo 2 – Padrão Arquitetural **27**

2.1 – Definição De Arquitetura De Software Corporativo **27**

2.2 – Principais Padrões Arquiteturais **29**

2.3 – Projeto Arquitetural **34**

Capitulo 3 – Benefícios De Um Projeto Arquitetural **36**

3.1 – Sistema Motivador **36**

3.2 – Benefícios **40**

Conclusão **41**

Referências **43**

INTRODUÇÃO

A evolução da computação dentro das corporações vem transformando os seus processos manuais e gerenciais em sistemas computacionais. Essa tarefa não é simples nem fácil de se realizar. Ela promove grandes desafios que arquitetos de software terão de enfrentar. Entretanto, é apresentado na arquitetura de software um conjunto de decisões de projeto que visa a sua correta elaboração e manutenção.

Provavelmente, um dos maiores desafios no projeto arquitetural é elaborar um modelo que contemple e permita que a aplicação suporte as frequentes alterações devido aos mais variados motivos. Uma nova lei fiscal, mudanças nas normas e metas internas, novas maneiras de atuar sobre um determinado problema ou simplesmente o fim da utilização de um processo etc.

Esses fatores provocam alterações diretas e indiretas nos sistemas corporativos e é comumente nesse ponto que surge o grande desafio: projetar um sistema para lidar com esse fato de forma a realizar tais mudanças em produção na certeza de que o mesmo continue a funcionar de forma correta e suporte novamente esse ciclo.

A comunidade de computação vem trabalhando com o objetivo de sugerir soluções para esses problemas e apresentar ideias eficazes e eficientes de forma a torná-las boas práticas arquiteturais. Dentro desse trabalho, destaca-se Fowler (2006), Gamma *et al*. (2000) que definem padrões arquiteturais e padrões de projeto, respectivamente. Convém mencionar que esses trabalhos criaram um estado da arte para os projetos arquiteturais.

Diante disso, este trabalho destina-se a analisar padrões de projeto e padrões arquiteturais bem como demonstrar seus reais benefícios no processo de manutenção dos sistemas de software utilizando de boas praticas para preservar a arquitetura planejada de forma a evitar a degradação da mesma.

Para desenhar uma aplicação corporativa com padrões arquiteturais e padrões de projeto é necessário descrever, entender e analisar onde e como eles são aplicados. Porém, antes de descrevê-los deve-se estudar os problemas que eles se propõem a resolver. A grande proposta desses é atuar sobre os conceitos da

orientação a objetos que tem sua base na componentização dos sistemas de forma a reutilizá-los.

> A arquitetura é subjetiva, uma compreensão do projeto de um sistema compartilhado pelos desenvolvedores experientes em um projeto. Esta compreensão compartilhada frequentemente se apresenta na forma dos componentes mais importantes do sistema e de como eles interagem. (FOWLER, 2006, p.24)

Entretanto, conceber sistemas orientados a objetos muitas vezes é visto apenas como a ocultação de métodos e funções em componentes para acessar a dados. Segundo Shalloway e Trott (2004), esse pensamento é limitado e incompleto, restrito em seu foco em como implementar objetos.

> O paradigma de orientação a objetos adotado nos anos 1980, no qual desenvolvedores de software eram levados simplesmente a encontrar substantivos nas declarações de requisitos e a transformá-los em objetos. Naquele modelo, o encapsulamento era definido como ocultação de dados, e os objetos, por sua vez, como coisas com dados e métodos utilizados para acessar tais dados, o que consistia em uma visão limitada e incompleta, restrita pelo seu foco em como implementar objetos. (SHALLOWAY; TROTT, 2004, p.30)

Gamma *et al*. (2000) lançaram um catálogo descrevendo padrões de projeto visando colaborar com a comunidade de desenvolvedores e estudiosos da problemática da correta utilização da orientação a objetos. Esse catálogo foi elaborado com base no estudo de vários arquitetos de software em sua constante busca por eficácia e eficiência.

> Um livro de padrões de projetos (*design patterns*) que descreve soluções simples para problemas específicos no projeto de software orientado a objetos. *Padrões de projeto* captura soluções que foram desenvolvidas e aperfeiçoadas ao longo do tempo. Daí, eles (os padrões de projetos) não servem os projetos que as pessoas tendem a gerar inicialmente. Eles refletem modelagens e recodificações, nunca relatadas, resultado dos esforços dos desenvolvedores por maior reutilização e flexibilidade em seus sistemas de software. *Padrões de projeto* capturam estas soluções em uma forma sucinta e facilmente aplicada. (GAMMA *et al*. ,2000, p.VIII)

O cuidado com esses padrões deve ocorrer também durante a manutenção do sistema e não apenas durante sua concepção. Como relatado por Ricardo Terra e Marco Túlio Valente:

> A utilização de padrões e a adoção de boas práticas são recomendações sempre realizadas em projetos de desenvolvimento de software. Contudo, com o decorrer do projeto, esses padrões tendem a se degradar fazendo com que os benefícios proporcionados por um projeto arquitetural (manutenibilidade, escalabilidade, portabilidade etc) sejam anulados. (TERRA; VALENTE, 2010, p.1)

A observação dos padrões arquiteturais e de projetos garante para o sistema de software escalabilidade, manutenabilidade, portabilidade etc. Desta forma, é recomendado que os arquitetos de software utilizem nos seus projetos esses padrões que já foram devidamente testados e documentados em vários projetos. Com isso, o arquiteto poderá se concentrar nos requisitos específicos do sistema que está modelando sem ter que “reinventar”soluções para problemas cujo soluções já são bem conhecidas.

CAPÍTULO 1 – PADRÕES DE PROJETO

Neste capítulo, é abordado um estudo sobre os principais padrões de projeto descrevendo seu nome, o problema motivador, a solução que ele propõe e as consequências de sua utilização. Na seção 1.1, é analisada a definição de padrões de projeto proposta por seus principais autores. Já nas seções 1.2, 1.3 e 1.4 são apresentados os grupos nos quais esses padrões são classificados. Em cada grupo é detalhado um relevante padrão de projeto e os demais são citados de forma sucinta.

Por fim, a seção 1.5 apresenta as limitações que os padrões de projeto têm na arquitetura do sistema como um todo, pois são a solução de um único problema. Essa seção apresenta também uma sucinta introdução aos padrões arquiteturais, mostrando como se propõem a resolver problemas arquiteturais em seu mais alto nível de forma a gerenciar os diversos padrões de projeto contemplados dentro da arquitetura de um sistema de software.

1.1. O que é um padrão de projeto

O termo padrão de projeto surge na engenharia civil onde sua ideia consiste em descrever um problema para um ambiente e reutilizar a solução. ALEXANDER[1] *et al.* (1977), citado por GAMMA *et al.* (2000), define que cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar essa solução infinitas vezes, sem nunca fazê-lo da mesma maneira.

Essa colocação também é válida para a computação quando nos referenciamos a padrões de projeto orientados a objetos. Um padrão de projeto para a computação consiste exatamente em definir uma solução dentro de um ambiente como resposta a um problema. Contudo, na computação, os padrões são colocados de tal forma que eles não são definidos por uma pessoa, mas sim, pela comunidade que observa soluções semelhantes propostas por diferentes arquitetos para os mesmos problemas.

[1] CHRISTOPHER, Alexander *et al.* **A Pattern Language**. New York: Oxford University Press, 1977.

Foi observando exatamente essas premissas que, em 1995, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides publicaram o catálogo de padrões de projeto. Eles observaram diversos projetos e notaram que as soluções propostas pelos arquitetos para os problemas eram semelhantes e visavam à flexibilidade e reutilização dos componentes dentro do sistema de software.

> Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Os padrões podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacentes. Em suma, ajudam um projetista a obter um projeto "certo" mais rápido. (GAMMA *et al.*, 2000, p.18)

Cada padrão de projeto descreve o nome do padrão, o problema, a solução e suas consequências. O nome deve ser uma referência ao problema que resolve e nos ajuda a criar um vocabulário específico facilitando assim a comunicação com outros arquitetos e a documentação dos sistemas. O problema descreve o contexto de quando aplicar um padrão. A solução descreve os elementos, seus relacionamentos, responsabilidades e colaboração. As consequências são as vantagens e desvantagens de utilizar os padrões.

> Um padrão de projeto nomeia, abstrai, e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizáveis. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve quando pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização. (GAMMA *et al.*, 2000, p.20)

Os padrões de projetos são organizados em grupos distintos que visam auxiliar no seu estudo, entendimento e na descoberta de novos. Eles são classificados nos grupos por dois critérios. O primeiro é a finalidade, que define o que o padrão faz. O padrão pode ter finalidade criacional, estrutural ou comportamental. O segundo critério é escopo. Ele especifica se o padrão se aplica a classe ou a objetos. A TAB. 1 apresenta os padrões segundo suas classificações.

Classificação dos padrões

		Propósito		
		Criacional	Estrutural	Comportamental
Escopo	Classe	*Factory Method*	*Adapter (classe)*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter (object)* *Bridge* *Composite* *Decorator* *Façade* *Flyweight* *Proxy*	*Chain of Responsability* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Tabela 1: Classificação dos padrões de projeto
Fonte: GAMMA *et al.*, 2000, p. 26.

1.2. Padrões de Criação

Segundo GAMMA *et al.* (2000), a família de padrões de criação tem por característica abstrair o processo de instanciação, colaborando para um sistema independente de como seus objetos são criados, compostos e representados. Para a classificação por classe, faz-se uso de herança para definir qual classe deve ser instanciada. Por sua vez, a classificação por objeto delega a criação para outro objeto. É importante ressaltar que os padrões de criação se tornam mais importantes à medida que o sistema evolui e passa a depender mais da composição dos objetos do que da herança.

Suprimir ou encapsular a forma que um objeto é instanciado fornece uma grande ferramenta para flexibilizar o sistema. Existem requisitos que exigem que um objeto seja instanciado apenas uma vez. Por exemplo, o *pool*[2] que controla as conexões de uma aplicação com o banco de dados, o objeto que representa a sessão do usuário, ou mesmo a criação de um objeto de forma dinâmica dependendo apenas das características que deva ter naquele momento.

Os padrões de projeto classificados como de criação são: *Abstract Factory*, *Singleton*, *Builder*, *Prototype* e *Factory Method*. Para um melhor

[2] O *pool* é o gerenciador de objetos de conexão com o banco de dados. (WIKI, 2010).

entendimento da classificação por criação, o padrão de projeto *Abstract Factory* é explicado de forma detalhada enquanto os outros são citados de forma sucinta.

O nome do padrão A*bstract Factory* demonstra exatamente o que ele faz uma fábrica abstrata de objetos. Tem por objetivo fornecer objetos concretos em tempo de execução dependendo da fábrica que é solicitada. Ele disponibiliza para o cliente uma abstração ou interface de suas fábricas que, por sua vez, retornam objetos concretos. Dessa maneira, o cliente não tem uma referência para o objeto concreto, mas sim, para as interfaces.

GAMMA *et al.* (2000) definem esse padrão como uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Dessa forma, o cliente pode trabalhar com os objetos concretos de forma dinâmica, alcançando assim, a flexibilidade do código. A FIG. 1 ilustra um diagrama que representa esse padrão.

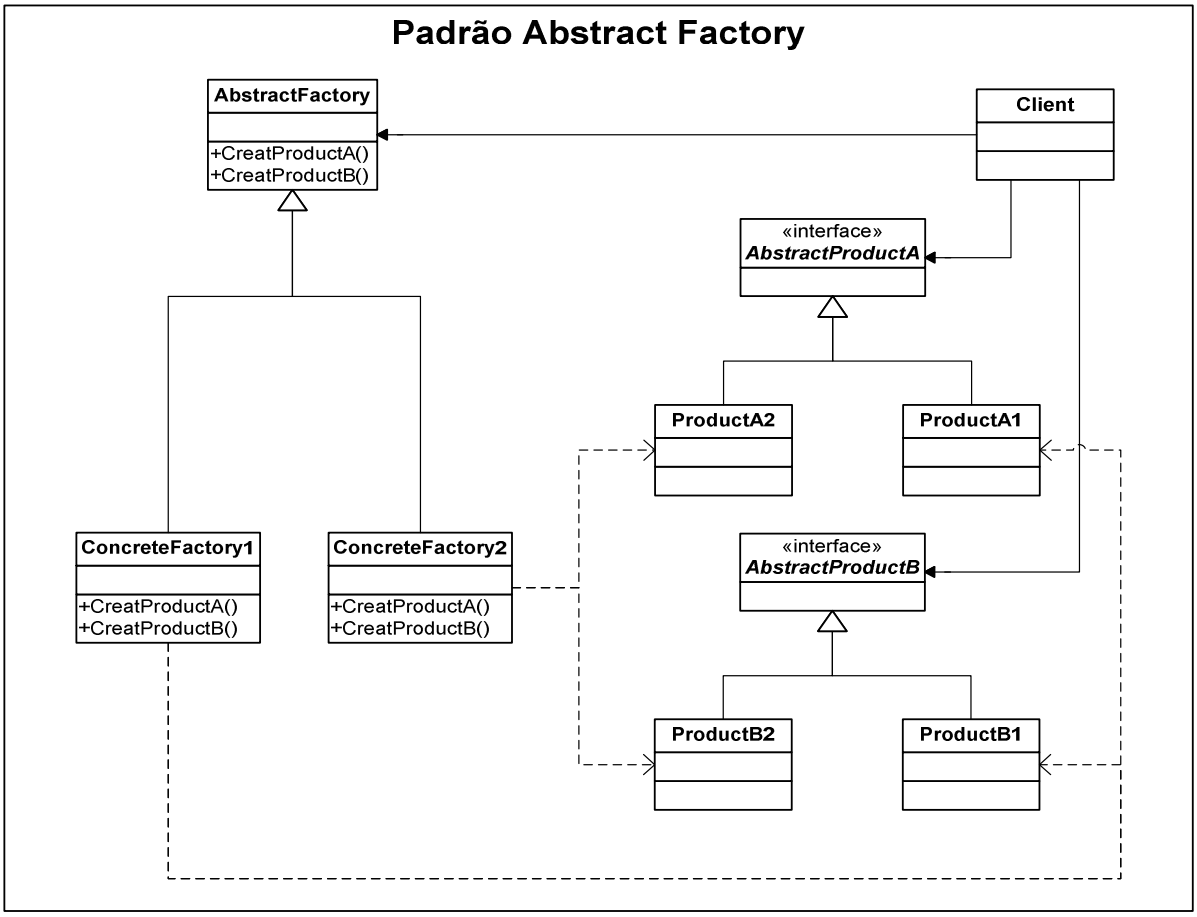

Figura 1: Padrão Abstract Factory
Fonte: GAMMA *et al.*, 2000, p.96

Como pode ser observado na FIG. 1 são definidos grupos de família de classes que implementam a mesma interface. Esses grupos são os produtos definidos nas famílias `produto A` e `produto B`. Define-se também, uma fábrica abstrata que contém apenas as assinaturas dos métodos que instanciam os produtos concretos. As fábricas concretas herdam da abstrata e implementam os métodos que instanciam os produtos concretos. Observa-se na FIG. 1 que as fábricas concretas têm conhecimento apenas dos produtos concretos instanciados por ela.

O cliente, por sua vez, não tem referência para os produtos ou fábricas concretas somente mantém referência para a fábrica abstrata e para as interfaces que definem as famílias de produtos. A grande vantagem da utilização desse padrão é que novos produtos ou fábricas podem ser definidos sem qualquer impacto no cliente. Isso torna o código flexível e garante coesão e acoplamento.

O código C# da LIST. 1 apresenta um exemplo da implementação desse padrão. Ele demonstra duas famílias de objetos uma representa os animais selvagens e a outra representa os continentes do planeta. O cliente tem referência para as duas interfaces que definem os animais e chama a fábrica abstrata que devolve os dois objetos concretos. Com essa abstração, o cliente não será impactado se um novo animal ou continente for adicionado ao projeto.

```
class AnimalMundo {
  private Herbivoro _herbivoro;
  private Carnivoro _carnivoro;

  public AnimalMundo(ContinenteFactory factory) {
    _carnivoro = factory.CriarCarnivoro();
    _herbivoro = factory.CriarHerbivoro();
  }

  public void MontaCadeiaAlimentar() {
    _carnivoro.Comer(_herbivoro);
  }
}

// A classe 'AbstractFactory'
abstract class ContinenteFactory {
  public abstract Herbivoro CriarHerbivoro();
  public abstract Carnivoro CriarCarnivoro();
}

class AfricaFactory : ContinenteFactory {
  public override Herbivoro CriarHerbivoro() {
    return new Gnu();
  }
```

```
  public override Carnivoro CriarCarnivoro() {
    return new Leao();
  }
}

class AmericaFactory : ContinenteFactory {
  public override Herbivoro CriarHerbivoro() {
    return new Bisao();
  }

  public override Carnivoro CriarCarnivoro() {
    return new Lobo();
  }
}

interface Herbivoro {}

class Gnu : Herbivoro {}

class Bisao : Herbivoro {}

interface Carnivoro {
  void Comer(Herbivoro h);
}

class Leao : Carnivoro {
  public void Comer(Herbivoro h) {
    Console.WriteLine(this.GetType().Name + " devora" + h.GetType().Name);
  }
}

class Lobo : Carnivoro{
  public void Comer(Herbivoro h) {
    Console.WriteLine(this.GetType().Name + " devora" + h.GetType().Name);
  }
}
```

Listagem 1: Código de exemplo do padrão de projeto Abstract Factory
Fonte: Do Autor

A LIST. 1 demonstra a abstração que os padrões de criação fornecem no momento de instanciar os objetos concretos. Nesse exemplo, pode-se observar que a classe `AnimalMundo` é o cliente portanto, essa tem referência apenas para a fábrica abstrata e para as interfaces que definem as famílias de objetos. A classe `ContinenteFactory` define apenas a assinatura dos métodos, transferindo a responsabilidade de instanciar as classes concretas para as classes que herdarem seu comportamento. Nesse casso, são as classes `AmericaFactory` e `AfricaFactory`. As duas interfaces `Carnivoro` e `Herbivoro` definem as famílias de classes de animais carnívoros e herbívoros, respectivamente.

Dentro dessa família de padrões, ainda temos o *Factory Method* que tem por objetivo segundo GAMMA *et al.* (2000) definir uma interface ou abstração para criar um objeto concreto, porém transferindo a responsabilidade de qual classe

instanciar para as subclasses. O *Factory Method* permite ainda a instanciação para subclasses.

O padrão *Builder* que visa, segundo GAMMA *et al.* (2000), separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações. Dessa mesma forma, o padrão *Prototype* tem por objetivo, segundo GAMMA *et al.* (2000), especificar os tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia desse protótipo.

Com o mesmo intuito de abstrair o processo instanciação de objetos que, segundo GAMMA *et al.* (2000), o padrão *Singleton* tem por objetivo, garantir que uma classe tenha somente uma instância e fornecer um ponto de acesso para garantir tal fato. Portanto, esses são os padrões de criação que fornecem uma ferramenta para abstrair a forma que os objetos são instanciados, garantindo assim uma forte flexibilidade para o sistema.

1.3. Padrões de Estruturais

Segundo GAMMA *et al.* (2000), os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores e complexas. Eles utilizam herança para compor interfaces ou implementações. O padrão é particularmente útil para fazer bibliotecas de classes ou, até mesmo, sistemas inteiros desenvolvidos independentemente trabalharem juntos. No lugar de compor interfaces ou implementações, os padrões estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades.

Os padrões de projeto classificados como estruturais são: *Adapter* (*class*), *Adapter* (*object*), *Bridge*, *Composite*, *Decorator*, *Façade*, *Flyweight* e *Proxy*. Para demonstrar o comportamento dos padrões classificados como estruturais, o padrão *Adapter* é apresentado de forma detalhada.

O padrão *Adapter* tem esse nome justamente para demonstrar a sua essência: um adaptador de classes. O seu objetivo é fornecer classes de adaptação permitindo ao sistema utilizar classes que ele não estava preparado ou fazer com que classes desenhadas inicialmente para não trabalharem juntas, mas que agora necessitam realizar um trabalho em conjunto.

GAMMA *et al.* (2000) definem esse padrão como conversor da interface de uma classe para outra interface que o cliente espera encontrar. O adaptador permite que classes com interfaces incompatíveis trabalhem juntas, proporcionando ao sistema uma melhor adaptação a mudanças. A FIG. 2 apresenta um diagrama que representa esse padrão.

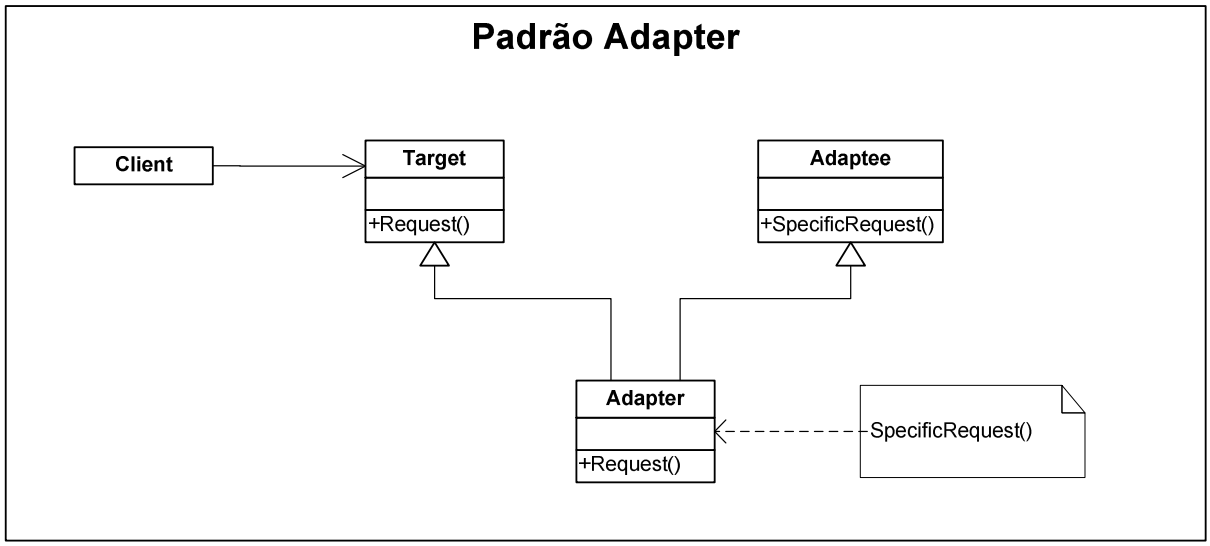

Figura 2: Padrão *Adapter*
Fonte: GAMMA *et al.*, 2000, p.142

Como pode ser observado na FIG. 2, a classe *Adapter* (adaptadora) realiza o acoplamento da classe T*arget* (alvo), chamada pelo cliente, à classe A*daptee* (adaptada). Isso implica que o cliente não sofrerá alteração na forma que chama a classe *Target*. Porém, quem realizará o que o cliente deseja é a classe A*daptee,* graças à classe *Adapter* que permite essa comunicação.

O código C# da LIST. 2 apresenta um exemplo da implementação desse padrão. Ele demonstra uma aplicação pratica para o padrão Adapter. Imagine que o sistema realize a divisão de dois números e retorne a resposta sem realizar o arredondamento. Todavia, agora existe um novo requisito que o sistema tem de se acoplar com outro que também realiza divisão, porém com o arredondamento.

```
//Adaptee
public class DivisaoSemArredondamento {
  public double FazDivisaoSemArredondamento(double a, double b) {
    return a / b;
  }
}
```

```
//Adaptador
public class Adaptador : IDivisaoComArredondamento {
  private DivisaoSemArredondamento _divisaoSemArredondamento = new
      DivisaoSemArredondamento();

  public int DivisaoComArredondamento(double a, double b) {
   double resposta = Math.Round(
_divisaoSemArredondamento.FazDivisaoSemArredondamento(a,b));
   return Convert.ToInt32(resposta);
  }
}


//Target
public interface IDivisaoComArredondamento {
  int DivisaoComArredondamento(double a, double b);
}
```

Listagem 2: Código de exemplo do padrão de projeto Adapter
Fonte: Do autor

A LIST. 2 demonstra o funcionamento do padrão *adapter*, note que a classe `Adaptador` realiza o acoplamento da classe `DivisaoSemArredondamento` com a classe `IDivisaoComArredondamento`. Esse fato implica que, o sistema que sabe apenas fazer a divisão sem arredondamento foi acoplado a um que necessita de uma divisão com arredondamento.

Dentro da família de padrões estruturais, ainda temos, o padrão *Bridge* que, segundo Gamma *et al.* (2000), tem como objetivo desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente. O *Composite* que compõe objetos em estrutura de árvore para representarem hierarquias parte-todo. Permitindo as classes tratarem de maneira uniforme objetos individuais e composição de objetos.

O padrão *Decorator* que visa, segundo Gamma *et al.* (2000), dinamicamente, agregar responsabilidades adicionais a um objeto. Eles fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades. Por sua vez, o padrão *Façade* fornece uma interface unificada para um conjunto de interfaces em um subsistema. Ele define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.

O padrão *Flyweight*, segundo Gamma *et al.* (2000), usa compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina. Por fim, Gamma et al. (2000) definem dentro desse grupo de família, o padrão *Proxy* que fornece um subconjunto ou marcadores da localização de outro objeto para controlar o acesso ao mesmo.

É dessa maneira que Gamma *et al*. (2000) definem os padrões estruturais que definem a forma como as classes e objetos devem ser estruturadas para produzir ao sistema maior flexibilidade e adaptação as mudanças e manutenções.

1.4. Padrões Comportamentais

Segundo GAMMA *et al.* (2000), os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidade entre objetos. Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles. Esses padrões caracterizam fluxos de controles difíceis de seguir em tempo de execução. Eles afastam o foco do fluxo de controle para permitir que você se concentre somente na maneira como os objetos são interconectados.

Os padrões comportamentais de classe utilizam herança para distribuir o comportamento entre classes. Os padrões comportamentais de objetos utilizam a composição de objetos no lugar da herança. Além disso, se preocupam com o encapsulamento de comportamento em um objeto e com a delegação de solicitações para ele.

Os padrões de projeto classificados como comportamentais são: *Interpreter*, *Template Method*, *Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy e Visitor*.

O padrão *Strategy* cria uma estratégia para alterar o comportamento do objeto em tempo de execução, encapsula o objeto concreto que está sendo utilizado e agrupa as famílias de objetos. O seu objetivo principal é fornecer flexibilidade ao sistema em tempo de execução, permitindo ter seu comportamento facilmente alterado.

GAMMA *et al.* (2000) descrevem o objetivo desse padrão como sendo definir uma família de algoritmos, encapsular cada uma delas e torná-las comunicativas. *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam. O diagrama a seguir, apresenta esse padrão.

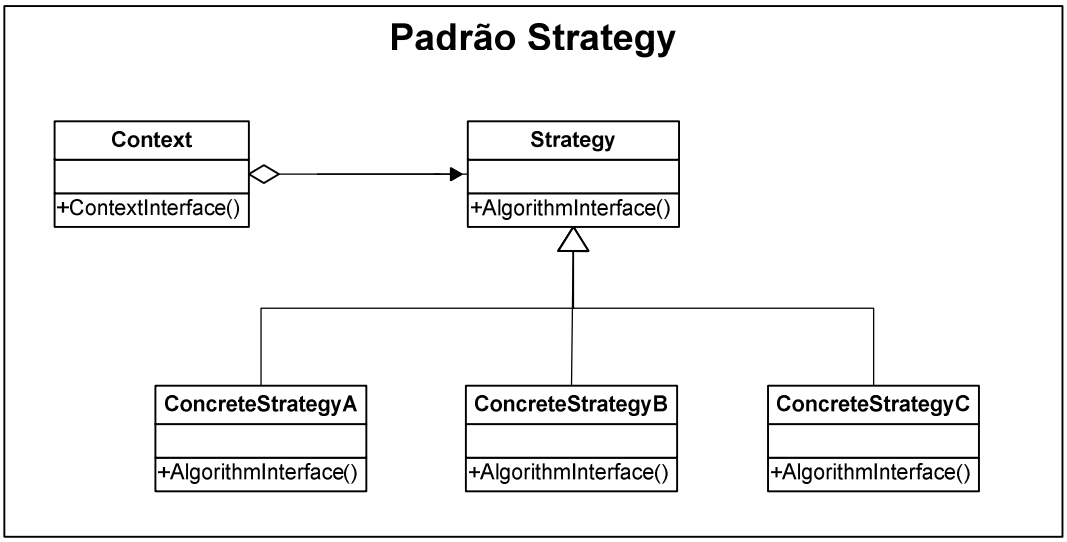

Figura 3: Padrão *Strategy*
Fonte: GAMMA *et al.*, 2000, p.294

Como pode ser observado na FIG. 3, as famílias de algoritmos relacionadas são agrupadas e seus comportamentos são separados de suas implementações. Esse fato pode ser observado uma vez que a classe `Context` tem apenas uma referência para a classe `Strategy` que é herdada pelas classes *strategy* concretas.

Essas classes *strategy* concretas encapsulam as famílias de comportamento e de classes concretas que devem ter esse comportamento. O padrão *Strategy* oferece uma resposta a hierarquia de classes que algumas vezes não é a melhor resposta para o comportamento dinâmico que objetos devam ter em tempo de execução.

O código C# da LIST. 3 apresenta um exemplo da implementação desse padrão. Ele demonstra uma família de classes de animais que seu comportamento é alterado em tempo de execução.

```
// A classe 'Strategy'
abstract class Animal {
  public abstract void FazerSomDoAnimal();
}

// A classe 'ConcreteStrategy'
class Porco : Animal {
  public override void FazerSomDoAnimal() {
    Console.WriteLine("Eu sou um porco - OINC OINC OINC");
  }
}
```

```
// A 'ConcreteStrategy' class
class Cachorro : Animal {
  public override void FazerSomDoAnimal() {
    Console.WriteLine("Eu sou um cachorro - AU AU AU");
  }
}

// A classe 'ConcreteStrategy'
class Gato : Animal {
  public override void FazerSomDoAnimal() {
    Console.WriteLine("Eu sou um gato - Miau Miau Miau");
  }
}

// A classe 'Context'
public class Context {
  private Animal _strategy;

  public Context(Animal strategy) {
    this._strategy = strategy;
  }

  public void ContextInterface() {
    _strategy.FazerSomDoAnimal();
  }
}
```

Listagem 3: Código de exemplo do padrão de projeto Strategy
Fonte: Do autor

A LIST. 3 demonstra o funcionamento do padrão *Strategy*, note que a classe `Context` tem referência apenas para a classe `Animal` que é abstrata e herdada pelas classes concretas que sabem como fazer o som do respectivo animal. Em tempo de execução, o animal pode adquirir diferentes comportamentos dependendo apenas da classe concreta animal que é passada como parâmetro para o método `FazerSomDoAnimal` na classe *`Context`*.

Dentro da família de padrões comportamentais, ainda temos, o padrão *Interpreter* que, segundo Gamma *et al.* (2000), tem como objetivo a parte de uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa representação para interpretar sentenças da linguagem. O *Template Method*, que define o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses. Ele permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.

O padrão *Chain of Responsibility* que, segundo Gamma *et al.* (2000), tem a responsabilidade de evitar o acoplamento do remetente ao seu receptor, ao dar a mais de um objeto a oportunidade e tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto o trate.

O padrão *Command* que, segundo Gamma *et al.* (2000), tem de encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro de solicitações e suportar operações que podem ser desfeitas.

O padrão *Iterator* que, segundo Gamma et al. (2000), fornece um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor a sua representação subjacentes. O padrão *Mediator* que define um objeto que encapsula a forma como um conjunto de objetos interage. Ele promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas intenções independentes.

O padrão *Memento* que, segundo Gamma et al. (2000), sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de maneira que o objeto possa ser restaurado para esse estado mais tarde. O padrão *Observer*, que define uma dependência um para muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.

O padrão *State* que, segundo Gamma *et al.* (2000), permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe. Finalmente, o padrão *Visitor* que representa uma operação a ser executada nos elementos de uma estrutura de objetos. Ele permite definir uma operação sem mudar as classes dos elementos sobre os quais opera.

Portanto, a família de padrões comportamentais, segundo Gamma et al. (2000), tem como característica o comportamento dos objetos, seja ele em tempo de execução ou não. Todos os padrões dessa família fornecem ferramentas para definir o comportamento dos objetos.

1.5. Limitações dos padrões de projeto

Como apresentado neste capítulo, os padrões de projeto buscam resolver problemas pontuais do sistema como instanciação, acoplamentos e comportamentos de classes. Eles não definem o sistema de forma abrangente, eles são visões localizadas. Em um sistema não é recomendado aplicar todos os padrões, já que muitos deles são concorrentes.

Os padrões de projeto podem ser vistos como uma caixa de ferramentas, onde cada ferramenta tem o seu objetivo específico. Porém, não é possível construir uma casa inteira apenas com ferramentas desse porte. Elas são muito úteis, mas inviáveis em certos casos. A mesma ideia se aplica na arquitetura de um sistema, utilizar apenas os padrões de projeto não garante a arquitetura do sistema como um todo e é, exatamente por isso que no próximo capítulo é apresentado o conceito de Padrões Arquiteturais.

CAPÍTULO 2 – PADRÃO ARQUITETURAL

Neste capítulo, é abordado um estudo sobre os padrões arquiteturais sua definição, utilização, benefícios e eventuais impactos nos sistemas de software. Este não tem por objetivo construir uma posição rígida e definitiva sobre o assunto, mas sim, princípios sólidos e consistentes para um estudo profundo do tema. Sendo assim, a seção 2.1 discute a definição de padrões arquiteturais. Já na seção 2.2, são abordados os dois padrões mais relevantes para a arquitetura: padrão de camadas e o MVC e, na seção 2.3, é apresentado um projeto arquitetural.

2.1. Definição de arquitetura de software corporativo

Segundo Fowler (2006), os padrões arquiteturais não são criações de uma pessoa, mas sim, a observação das coisas que ocorrem na prática. Fowler, assim como Gamma, utilizam como referência a definição de Alexander *et al.* como ponto de partida para a definição de padrões. Porém, diferente dos padrões de projeto, os arquiteturais são mais abrangentes para o sistema.

Fowler (2006) fornece uma ótima demonstração desse fato na introdução do seu livro quando fala que a arquitetura é subjetiva, uma compreensão do projeto de um sistema compartilhada pelos desenvolvedores experientes em um projeto. É exatamente nesse momento que ele inicia a formação de uma ideia mais concreta sobre o assunto quando afirma “Esta compreensão compartilhada frequentemente se apresenta na forma dos componentes mais importantes do sistema e de como eles interagem.” (FOWLER, 2006, p. 24).

A definição de arquitetura de software é tão genérica e imprecisa que um dos institutos mais renomados mundialmente sobre o assunto, o *Software Engineering Institute* de *Carnegie Mellon,* tem em seu portal não uma, mas várias definições para arquitetura de software. Alguma delas são:

> A arquitetura de software de um sistema consiste de componentes / módulos que fazem o sistema, suas relações e a interação para atender aos requisitos do mesmo. A arquitetura foca em como implementar atributos não funcionais e funcionais. (Ranbir Singh – Lead Consultant, Capgemini FS GBU, Pune, Maharashtra, India) (Tradução nossa)

> Arquitetura de software é definida como um estilo que é provado cientificamente e adotado pela disciplina de engenharia, com a qual um software é desenvolvido tanto para sustentar quanto para adotar a

> necessidade de crescimento da indústria de tempos em tempos. (Phaniraj Adabala – Systens Manager, Prasad Film Laboratories, Chennai, TN India) (Tradução nossa)

> Arquitetura de software é definida como a representação de uma linguagem independente de um dado sistema de software. Ela representa o esqueleto do sistema de software, enfatizando a definição clara da estrutura, comunicação e inter-relacionamento do corpo dos componentes que preenchem o propósito de um dado software. Os componentes da arquitetura devem expor os protocolos dos componentes apenas para seus clientes. A arquitetura de software deve implementar uma separação clara de preocupações em todo núcleo observado e não observado do comportamento de um sistema de software. A arquitetura balanceia toda interoperabilidade desse comportamento do núcleo para preencher as necessidades dos sistemas de software dos usuários. A arquitetura de software deve suportar ambos componentes de configuração estática e dinâmica através de uma série de interfaces bem documentadas ou contratos conhecidos pelos clientes de um componente de um dado sistema. A arquitetura de software deve suportar a manutenabilidade, escalabilidade, portabilidade e reusabilidade. Apenas para citar alguns poucos. (Ebenezer Adegbile – Consultant, Self Employed, London, England) (Nossa tradução)

> A arquitetura de software está alinhando os componentes de software com a colaboração e integração apropriadas para se sustentar no ambiente de negócio. (Vijaya Agarwal – Associate Consultant, ICFAI, Hyderabad, India) (Nossa tradução)

Trabalhar com a compreensão de arquitetura de software como a interação dos principais componentes é o princípio do pensamento arquitetural. Para os padrões de projeto, o principal pensamento é sempre flexibilidade. Para isso, nunca deve-se programar para classes concretas, mas sempre para interfaces ou classes abstratas.

Na arquitetura, o pensamento é sempre componentizar ou modularizar o sistema, de tal forma que um sistema grande e complexo se transforme em um pequeno e coeso, preservando a interação entre seus elementos ou componentes. É essa abordagem utilizada por Terra (2010) quando define que a arquitetura de software é normalmente definida como um conjunto de decisões de projeto que inclui como os sistemas são estruturados em componentes e as restrições sobre como tais componentes devem interagir.

Assim, podemos entender que a arquitetura de software não engloba apenas um item ou uma definição, mas sim, um conjunto de diversos padrões e boas praticas visando a manutenibilidade, escalabilidade e portabilidade dos sistemas de software. Uma definição largamente aceita sobre padrões é que uma solução consiste em uma solução padrão ou compartilhada para um problema recorrente.

2.2. Principais padrões Arquiteturais

Fowler, em seu livro Padrões de Arquitetura de Aplicações Corporativas apresenta vários padrões arquiteturais que são extremamente úteis para qualquer arquiteto, projetista ou analista de software. Porém, ele não foi o primeiro arquiteto de software a observar essas soluções recorrentes para a definição de uma arquitetura.

Em 1979, Reenskaug trabalhando para Xeros com a linguagem *Smalltalk*, publica o artigo *Applications Programming in Smalltalk-80: How to use Model-View-Controller,* Considerado um dos primeiros e mais importantes artigos publicados para a área de arquitetura de software.

Outro padrão largamente utilizado na elaboração dos sistemas de software é o padrão de camadas. Dividir a responsabilidade do sistema em domínios e colocar cada um em uma camada é uma tarefa relativamente simples e pode oferecer uma série de benefícios. Segundo FOWLER (2006), podemos compreender uma única camada como um todo coerente. Pode-se substituir camadas por implementações alternativas dos mesmos serviços básicos.

> Ao pensar em sistema em termos de camadas, você imagina os subsistemas principais no software dispostos de forma parecida com camadas de um bolo, em que cada camada repousa sobre uma camada mais baixa. Nesse esquema, a camada mais baixa ignora a existência da camada mais alta. Além disso, cada camada normalmente esconde suas camadas mais baixas das camadas acima, então a camada 4 usa os serviços da camada 3, a qual usa os serviços da camada 2, mas a camada 4 ignora a existência da camada 2. (FOWLER, 2006, p.37)

Normalmente nas literaturas, as camadas utilizadas são A*presentação*, A*plicação*, *Negócio* e P*ersistência*. Conforme ilustrado na FIG. 4.

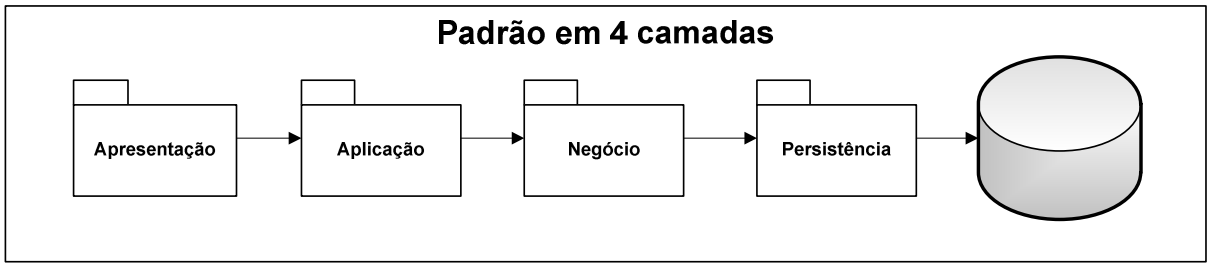

Figura 4: Padrão em quatro camadas.
Fonte: Do autor

A camada de *Apresentação* é responsável por agrupar todas as interfaces do sistema com o usuário. Essas interfaces podem ser telas do sistema, relatórios, gráficos, mapas etc. A camada de *Aplicação* é responsável por agrupar as classes que recebem os dados da interface e repassa para as camadas de *Negócio* e *Persistência*, essa trabalha como um orquestrador dos dados.

A camada de *Negócio* é responsável por agrupar as classes que compõem o domínio do negócio. Por exemplo, para um sistema de uma empresa de aluguel de veículos as classes de negócio podem ser cliente, veículo, agência, pagamento etc. A camada de *Persistência* é responsável, como demonstrado na FIG. 4, por agrupar todas as classes que tratam da leitura, inserção, atualização e exclusão de informações no banco de dados.

Um ponto relevante sobre a arquitetura em camadas é que a camada inferior não tem conhecimento sobre as superiores. Dessa forma, pode-se alterar qualquer camada do sistema sem afetar outras. Um exemplo clássico para esse comportamento é migrar uma aplicação *desktop* para *web* ou *mobile*. Uma solução arquitetural é alterar apenas a camada de *Apresentação*. Assim, o sistema pode ser migrado sem maiores impactos.

Um dos modelos práticos para aplicação em camadas é o *three tier* ou três camadas. Onde três camadas e não mais quatro são suficiente para representar o sistema. FOWLER (2006) define essas três camadas como sendo camada de apresentação, de dados e lógica de domínio. Conforme visto na FIG. 5.

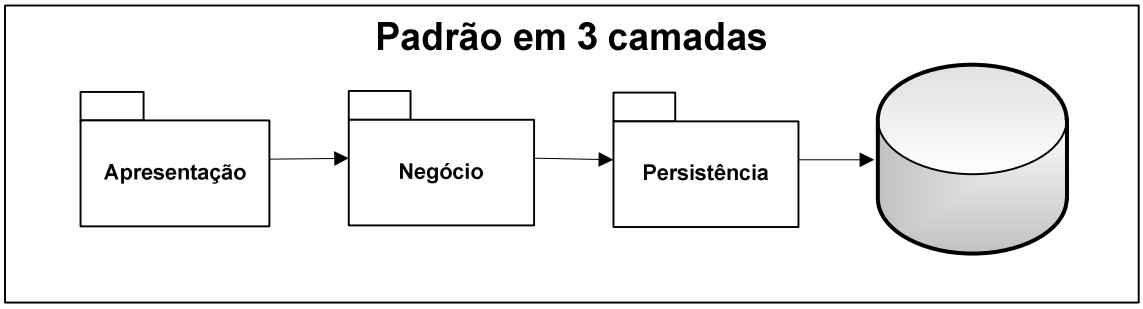

Figura 5: Padrão em três camadas.
Fonte: Do autor

> A lógica de apresentação diz respeito a como tratar a integração entre o usuário e o software. Isso pode ser tão simples quanto uma linha de comando ou um menu baseado em texto, porém, hoje é mais provável que seja uma interface gráfica em um cliente rico ou um navegador com interface baseada em HTML. As responsabilidades primárias da camada de apresentação são exibir informações para o usuário e traduzir comandos do

> usuário em ações sobre o domínio e a camada de dados. (FOWLER, 2006, p.40)
>
> A lógica de camada de dados diz respeito à comunicação entre outros sistemas que executam tarefas no interesse da aplicação. Esses podem ser monitores de transações, outras aplicações, sistemas de mensagens e assim por diante. Para a maioria das aplicações corporativas, a maior parte da lógica de dados é um banco de dados responsável, antes de mais nada, pelo armazenamento de dados persistentes. (FOWLER, 2006, p.40)
>
> O resto é a lógica de domínio, também chamada de lógica de negócio. Este é o trabalho que esta aplicação tem de fazer para o domínio com o qual você esta trabalhando. Envolve cálculos baseados nas entradas e em dados armazenados, validações de quaisquer dados provenientes da camada de apresentação e a compreensão exata de qual lógica de dados executar, dependendo dos comandos recebidos da apresentação. (FOWLER, 2006, p.40)

Como visto na FIG. 5 e nas definições de Fowler, a arquitetura em três camadas é bastante similar a de quatro, contudo, a retirada de uma camada proporciona uma facilidade de integração com o modelo *Model-View-Controller* proposto por *Reenskaug*.

O modelo *Model-View-Controller* tem por objetivo dividir a aplicação em três camadas, trazendo respostas para o novo paradigma de aplicações existentes. Antigamente, as aplicações eram pesadas apenas para rodar em um único computador e a sua interface com o usuário era mínima e pobre. Às vezes, esses sistemas eram projetados apenas para processar grandes quantidades de dados, como é o caso dos sistemas que processavam arquivos *batch* (arquivos de lote).

Todavia, essa realidade foi sendo alterada principalmente pelo advento e popularização da web. As corporações foram uma das pioneiras no desejo de automatizar seus sistemas, disponibilizá-los dentro da rede interna e, na própria web. Atualmente, não vemos problema algum em acessar o sistema da empresa de casa ou realizarmos compras pela Internet.

Atualmente arquiteto, projetista ou analista pensar na arquitetura desses sistemas da mesma maneira que se pensava a arquitetura dos sistemas antigos que rodavam em um único computador e apenas processava dados é inviável. Hoje em dia, os clientes querem executar sistemas em vários computadores, com dados atualizados, tendo interfaces ricas. Já os empresários querem sistemas acopláveis., isso é, sistemas que comunicam com outros já existentes.

O padrão arquitetural *Model-View-Controller* ou MVC apresenta uma excelente resposta para esses problemas. Cada camada é teoricamente independente uma da outra produzindo um isolamento da lógica do negócio com a

interface com o usuário. Isso permite que uma camada seja alterada sem impactar a outra. Em suma, o modelo encapsula os dados da aplicação, o controle processa a entrada do usuário e a interface exibe dados para o usuário.

De forma detalhada a camada *Model* representa o domínio específico da aplicação. O estado dela em um determinado momento para ser mais exato. Por exemplo, em um sistema de uma escola as classes professor, aluno, curso etc estão no domínio dessa aplicação. Normalmente, os arquitetos e analistas iniciantes confundem o *Model* do padrão MVC com a camada negócio do padrão em camadas. Mesmo com a similaridade, não são a mesma coisa. O *Model* é a representação das entidades no banco de dados, dessa forma, as classes que o definem contem apenas atributos e propriedades *get* e *set*. Já as a camada de negócio é o local onde as regras do sistema deve ser colocada e definida, dessa forma, as classes que a definem são compostas em sua maioria por vários métodos que contem as regras de negócio.

A *View* exibe os dados para o usuário da aplicação. Entretanto, ela exibe os dados do *Model* naquele momento para a interação do usuário. Voltando ao nosso exemplo da escola a *View* representa os dados do aluno, professor, curso etc daquele momento no *Model* na aplicação.

O *Controller* por sua vez é responsável por processar e responder aos eventos e ações do usuário na *View* e solicita alterações no *Model.* Voltando novamente ao nosso exemplo da escola, se um usuário alterar informações de um aluno na *View* é responsabilidade do *Controller* informar essa alteração para o *Model*.

A FIG. 6 demonstra como é feita a comunicação entre os componentes *Model, View e Controller*. Note que o *View* tem certo conhecimento do *Model*, mas não sabe nada sobre quem responde a suas ações, nesse caso o *Controller*. Esse por sua vez, conhece tanto a *View* com o *Model*, pois intercepta as ações/eventos de um e atualiza os dados em outro. Já o *Model* não tem conhecimento de quem utiliza os seus dados, simplesmente sabe que tem os dados conhece seu estado atual.

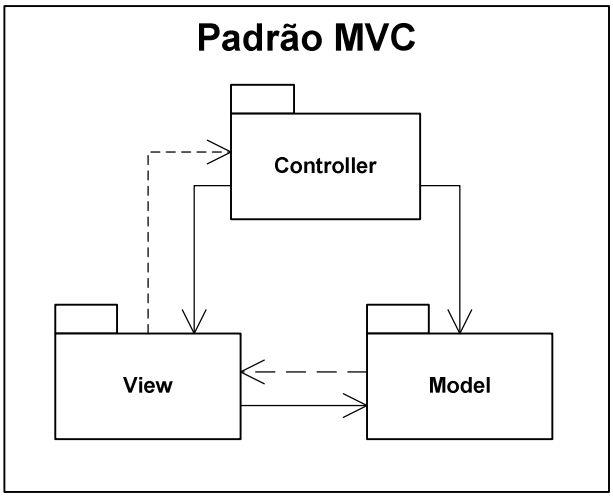

Figura 6: Padrão MVC.
Fonte: do autor

Uma questão bastante relevante sobre o modelo MVC é que o mesmo não prevê uma camada de persistência de dados. Por esse motivo muitos arquitetos normalmente fazem uma fusão desses dois padrões arquiteturais visando à persistência em bancos de dados. Alguns arquitetos e analistas optam por acrescentar mais uma camada, outros concordam que essa responsabilidade é do modelo, colocando apenas as classes que sabem como ir ao banco de dados nessa camada.

Esse ponto de vista varia entre os arquitetos e analistas. Por exemplo, dependendo da linguagem que se trabalha do sistema que esteja sendo desenvolvido, as camadas vão mudar de responsabilidade e, até mesmo, de nomes. Aplicações corporativas têm camadas diferentes das aplicações distribuídas. Mas, o pensamento do arquiteto, projetista, analisa de sistemas deve sempre ser o de separar a responsabilidade do sistema em camadas/blocos visando sua manutenibilidade.

Entretanto, comvém ressaltar que o padrão arquitetural MVC (*Model-View-Controller*) determina como vai ocorrer a comunicação dos componentes, níveis ou camadas da aplicação. Já o padrão arquitetural de três, quatro ou N camadas define como agrupar os componentes da aplicação.

2.3. Projeto Arquitetural

Doutor em Engenharia Elétrica e professor Jacques Philippe Sauvé's da Universidade Federal de Campina Grande define que um projeto arquitetural são decisões estratégicas que terão consequências profundas e que essas decisões são tomadas em alto nível.

Doutor em Ciência da Computação e professor Paulo Caetano da Universidade Federal da Bahia define que um projeto arquitetural consiste em se buscar uma solução para uma tecnologia específica para a satisfação dos requisitos levantados. Determinam-se quais processos executarão, em quais processadores, onde os dados serão armazenados e qual a comunicação necessária.

Doutores Ricardo Terra e Marco Túlio da Universidade Federal de Minas Gerais definem que uma arquitetura engloba diversos padrões e boas práticas arquiteturais. Contudo, com o decorrer do projeto, esses padrões tendem a se degradar fazendo com que os benefícios proporcionados por um projeto arquitetural sejam perdidos.

Essas definições são apenas pequenas amostras do que vem a ser um projeto arquitetural de um sistema de software. Até agora, viu-se que o os padrões de projeto buscam soluções para problemas específicos dentro do desenvolvimento de sistemas e que a arquitetura de software visa a componentização e comunicação desses componentes do sistema de software. Mas, ainda não tem-se um ponto que demonstre um sistema como um todo, representando os requisitos funcionais e não funcionais do projeto de software.

É nesse ponto que a Arquitetura de software tange a Engenharia de Software, o Projeto Arquitetural é o elo entre as duas. Da mesma maneira que a arquitetura de software não tem uma única definição, o projeto arquitetural também não. Para alguns arquitetos e analistas, essa definição está mais próxima do desenho arquitetural do sistema e pode ser construída com base na documentação UML e nos processos como RUP, CMMI, MPS.Br etc. Utilizados no desenvolvimento do sistema de software. Para outros, está mais próxima da arquitetura de software em si, não sendo necessário a vinculação com processos de software.

No entanto, esta discussão e estudo não trata a definição teórica de projeto arquitetural. Mas sim, um ponto de vista mais prático fornecendo respostas a questões como: Para que é utilizado? Quais os benefícios que um projeto

arquitetural pode trazer para o desenvolvimento e manutenabillidade de um sistema? Por que não é um processo simples, bastando definir as camadas e as responsabilidades. Tais questões e suas respectivas respostas compõem o assunto do próximo capítulo.

CAPÍTULO 3 – BENEFÍCIOS DE UM PROJETO ARQUITETURAL

Nos capítulos anteriores, foram apresentados os padrões de projeto e os fundamentos dos padrões arquiteturais, com o objetivo de discutir neste capítulo os benefícios da aplicação desses padrões na manutenção de sistemas de software, por meio de situações práticas experimentadas no desenvolvimento de sistemas. Por exemplo, quais impactos terá a aplicação se não for separada em camadas? E a principal pergunta desse trabalho: Aplicar padrões no projeto de software realmente interfere na sua manutenabilidade?

Segundo a Engenharia de Software, os custos financeiros mais elevados de um sistema de software não estão nas suas fases de iniciação, elaboração, construção ou transição. Mas sim, na sua fase de alteração ou manutenção. Assim, como pode-se realizar manutenção em um sistema visando diminuir esse custo?

Realizar manutenção em sistemas não provoca impacto apenas financeiro. Segundo Terra e Valente (2010), esse fato impacta também na arquitetura que tende a se degradar, fazendo com que seus benefícios sejam perdidos. Como então tratar o problema recorrente no desenvolvimento de software: “Os requisitos de um software mudam e um sistema em ambiente de produção certamente sofrera alterações”? Dessa forma, este capítulo discute a importância de aplicar os padrões no desenvolvimento de sistemas de software.

3.1. Sistema Motivador

Visando aplicar as boas práticas arquiteturais, fazendo uso dos padrões de projeto e padrões arquiteturais, utiliza-se a idealização de um sistema chamado `TMTutismo` que realiza pacotes turísticos para todo o mundo. A principal funcionalidade será a criação de pacotes turísticos que contemplam o destino da viagem, a passagem, a hospedagem e algumas opções que devem ser fechadas na contração do pacote como, por exemplo, aluguel de veículos, passeios e atividades na região.

A arquitetura do sistema será baseada no padrão arquitetural de camadas, nesse caso quatro camadas: *Apresentação*, *Negócio*, *Persistência* e *Modelo*, conforme ilustrado na FIG. 7.

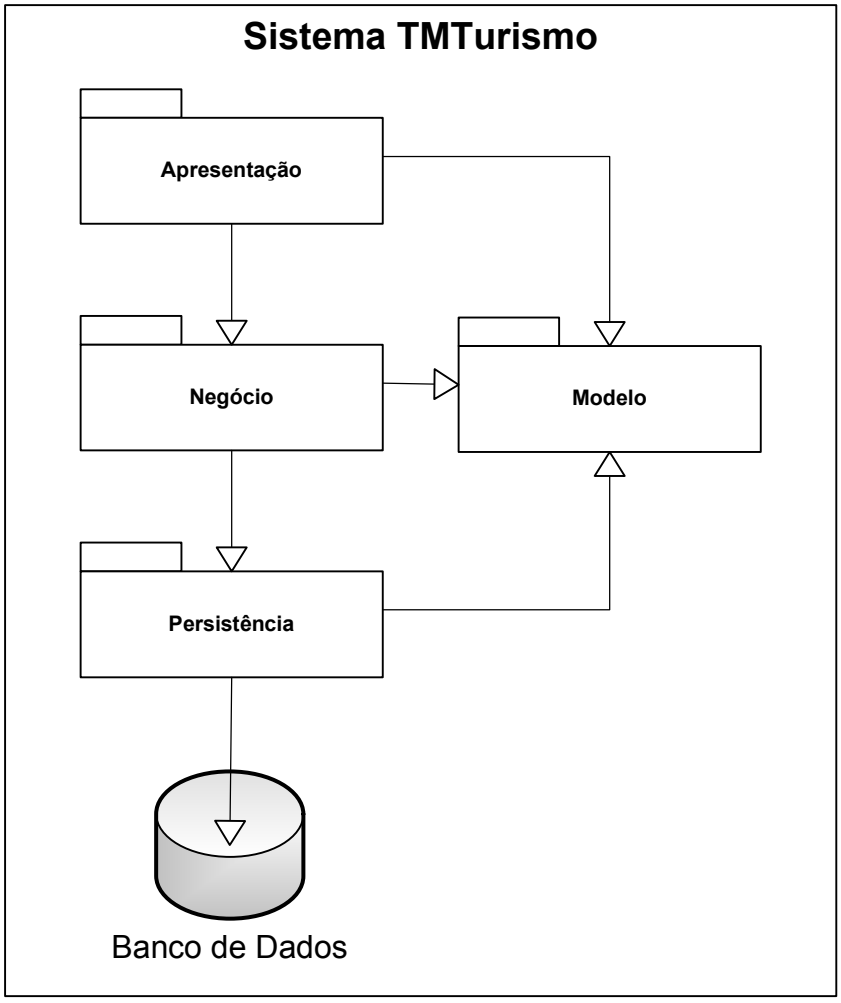

Figura 7: Arquitetura do Sistema `TMTurismo`
Fonte: Do autor

A FIG. 7 demonstra como é o relacionamento das camadas do sistema. É visto que a camada de *Apresentação* tem conhecimento da camada de *Negócio* e *Modelo*. A camada de *Negócio* tem conhecimento da camada de *Persistência* e *Modelo*. A camada de *Persistência* tem conhecimento da camada de *Modelo* que por sua vez, não tem conhecimento de nenhuma camada do sistema, isto é apenas é conhecida por todas as outras.

A camada de *Apresentação* tem todas as telas do sistema, relatórios, gráficos, mapas, imagens e todos os mecanismos que realizem a interface com o usuário. No desenvolvimento dessas interfaces será utilizado o Ajax Asp.Net[3]. A camada de *Negócio* tem toda a regra de negócio do sistema. A camada de *Persistência* tem as classes que sabem como persistir e recuperar os dados do

[3] http://www.asp.net/ajax

banco, na implementação dessa camada será utilizado o *framework* ADO.Net Entity Framework[4]. E a camada de *Modelo* tem as classes que representam o modelo do negócio.

Dessa forma, obtêm-se o agrupamento das classes nas camadas específicas. Com o intuito de garantir-se a restrição de acessos das camadas, proporcionando assim a integridade do sistema, a camada de *Apresentação* só pode acessar a camada de *Negócio* ou *Modelo*, a camada de *Negócio* somente acessa a camada de *Persistência* ou a camada do *Modelo*, a camada de *Persistência* somente acessa a camada do *Modelo*, a camada do *Modelo* somente é acessada pelas outras camadas. Esse fato é ilustrado na FIG. 7.

Visando garantir uma melhor arquitetura para o sistema, buscando sua flexibilidade para as alterações e correto reaproveitamento de código, esse deve contemplar a aplicação dos padrões de projeto, já que sabe-se dos seguintes requisitos.

O sistema `TMTurismo` deve ser flexível o suficiente para o atendente modelar um pacote turístico de acordo com a preferência do cliente. Esse fato implica que o cliente vai informar qual o destino quer ir, o tipo de passagem, a hospedagem e as opções de veículo e passeios juntamente com atividades desejadas. Sabe-se também que a empresa `TMTurismo` solicitará a alteração dos itens que formam o pacote de turismo sempre que houver a necessidade de tal fato.

Munidos dessas informações, é definido que o ponto crítico do sistema é a camada de *Negócio*, pois essa sofrerá alterações constantemente. Sendo assim, deve-se aplicar os padrões de projeto principalmente nesse ponto. Um dos padrões de projetos aplicados nessa camada será o *Strategy* para a geração dos pacotes de turismo. Conforme ilustrado na figura 8.

[4] http://msdn.microsoft.com/en-us/data/ef.aspx

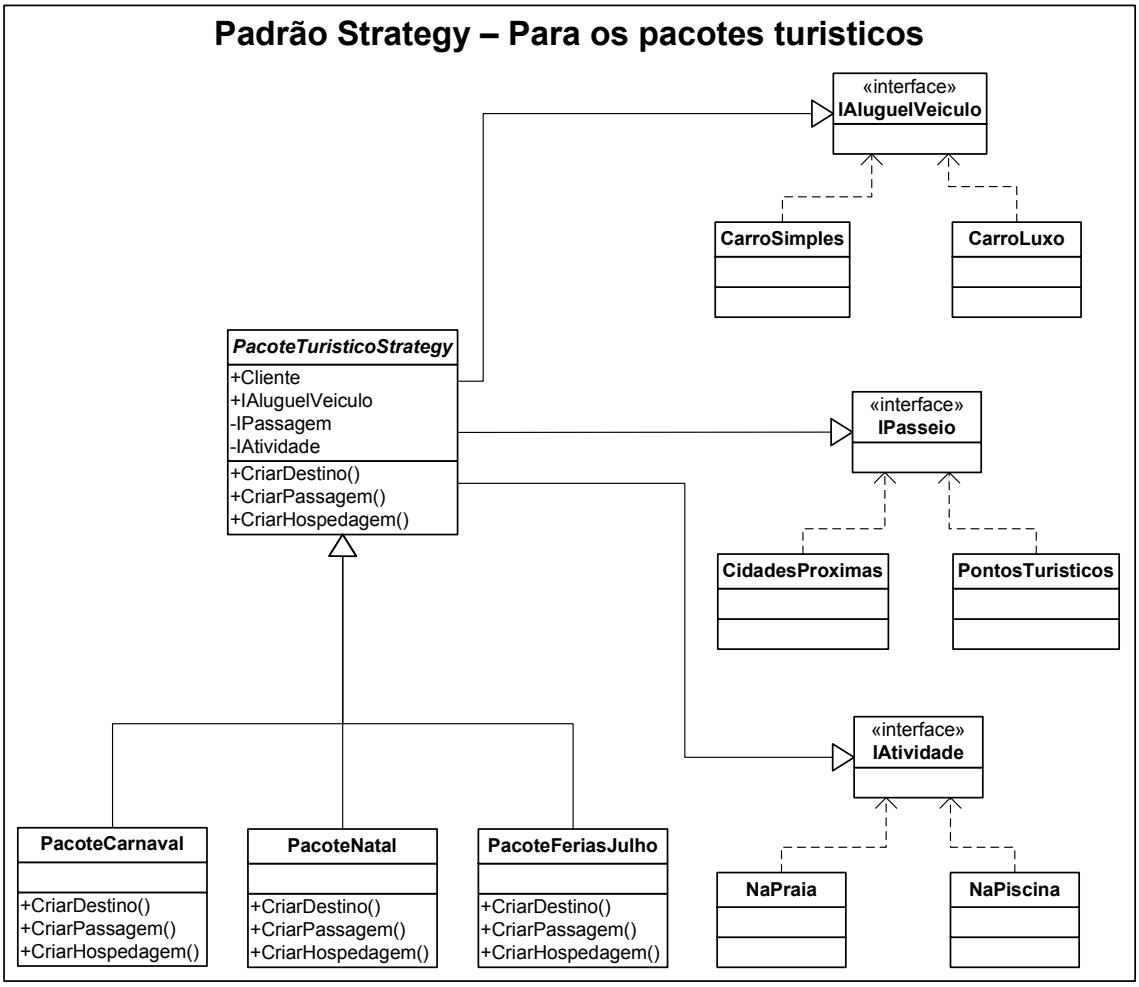

Figura 8: Padrão Strategy na camada de negócio do sistema MTTurismo
Fonte: Do autor

O padrão *Strategy* deve ser implementado na camada de negócio com o intuito de proporcionar alterações nos pacotes turísticos de forma organizada e planejada no código fonte, uma vez que se novos pacotes forem criados ou novos comportamentos adicionados, a camada de negócio estará preparada. Basta apenas herdar da classe abstrata `PacoteTuristicoStrategy` (no caso da criação de novos pacote) ou criar uma interface com sua família de comportamento adicionando a sua referência na classe `PacoteTuristicoStrategy`.

3.2. Benefícios

Os benefícios alcançados com a arquitetura do sistema em camadas é que cada alteração do sistema pode ser realizada em pontos específicos. Por exemplo, existe o novo requisito de colocar o sistema disponível para componentes *mobiles*. A

alteração do sistema para atender a esse novo requisito ocorrerá em toda a camada de apresentação, mas as outras camadas não sofrerão alteração alguma.

Outro requisito para alteração do sistema poderia ser a utilização de uma nova versão de sistema SGBD (Sistema Gerenciamento de Banco de Dados). Não será mais utilizado o Oracle ou o Sybase, mas agora deve ser utilizado apenas o SQL Server. Apenas a camada de persistência deve sofrer alterações para atender a essa demanda.

Note que desenhando o sistema com os padrões arquiteturais, as alterações que vão ocorrer podem ser atendidas sem maiores impactos. Isso ocorrera pois, o sistema foi projetado de forma a permitir a sua flexibilidade. Isso proporciona uma manutenção mais rápida, segura e eficaz, uma vez que já se sabe onde realizar a alteração do sistema. Alem disso, os testes ficam mais simples, pois pode-se realizar o teste apenas na camada que foi alterada economizando assim tempo, utilização de recurso e dinheiro.

Esse fato não ocorreria se o sistema não fosse desenhado em camadas. Pois, neste caso, não seria possível definir se as interfaces contêm as regras de negócios ou acessam diretamente o banco de dados. Para trocar o SGBD, por exemplo, de Oracle para SQL Server todo o sistema deveria ser analisado para verificar onde é realizado o acesso ao banco de dados. Ressaltando que, no final das alterações, todo o sistema devera ser testado. Neste caso a alocação de recursos e o gasto de tempo e dinheiro será bem mais elevado.

A utilização do padrão arquitetural de camadas não garante para o sistema uma flexibilidade. Para isso, dessa forma é necessário também se aplicar alguns dos padrões de projeto visão a reutilização de código.

CONCLUSÃO

Desenhar e construir aplicações corporativas é uma ciência, e como tal, deve-se obedecer a princípios, normas, leis e recomendações. Um sistema de software corporativo deve ser pensado não apenas na sua elaboração. Alterações nesse sistema certamente ocorrerão e esse deve estar preparado para tal fato. A aplicação dos padrões arquiteturais e dos padrões de projeto garantem ao sistema grandes benefícios no processo de manutenção.

Pode-se observar tal fato no livro de GAMMA *et al.* (2000) que é utilizado como principal referencia no primeiro capitulo desta. Os padrões de projeto são um recurso extremamente útil na construção e manutenção de sistemas de software pois, promovem o mesmo vocabulário entre a equipe, facilitando assim a comunicação e suas implementações já foram testadas várias vezes, o que garante a eficácia e eficiência de sua implementação.

Somente a aplicação dos padrões de projeto não garante ao sistema a sua correta arquitetura, uma vez que esses propõem soluções para problemas pontuais e não de toda a aplicação. Sendo assim, se faz necessário aplicar outros recursos da alcançar-se o estado da arte da arquitetura bem planejada.

Com o intuito de solucionar a lacuna existente para a arquitetura do projeto foi que Fowler escreveu o livro sobre os padrões arquiteturais. Esse é utilizado como base do segundo capítulo desta monografia e apresenta as soluções que Fowler propõe para os problemas arquiteturais. Essas, não são pontuais como os padrões de projeto, mas abrangentes. Os padrões arquiteturais propõem soluções gerais para a arquitetura do projeto. Tal fato também contribui para a melhor comunicação da equipe, assim como, para a organização do projeto.

O terceiro capítulo desta monografia propõe um sistema motivado que demonstra como os padrões de projeto e os padrões arquiteturais podem se aplicados de forma prática para atingir os objetivos propostos por seus autores e apresentados nos dois primeiros capítulos.

Portanto, pode-se concluir depois desse estudo que desenvolver aplicações corporativas sem o correto planejamento e utilização dos padrões somente levará o sistema a um completo e total caos durante o seu processo de sua manutenção. Onerando assim, gastos com recursos alocados, testes, duplicação de código etc.

Sem mencionar que, o esforço para se reorganizar o código na etapa de manutenção é maior do que na etapa de elaboração.

Pode-se concluir que a utilização de padrões arquiteturais e de projeto que já foram testados e homologados mais de uma centena de vezes é uma solução altamente recomendada durante a fase de elaboração de um sistema de software. Se esses padrões forem aplicados, certamente a manutenibilida, escalabilidade e portabilidade desse sistema estará preservada.

REFERÊNCIAS

Community Software Architecture Definition. **Software Engineering Institute**. Disponível em: <http://www.sei.cmu.edu/architecture/start/community.cfm>. Acesso em: 10 nov. 2010.

FOWLER, Martin *et al.*, **Padrões de arquitetura de aplicações corporativas**. São Paulo: Bookman, 2006.

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

MVC. Disponível em: < http://pt.wikipedia.org/wiki/MVC>. Acesso em: 12 nov. 2010.

NETO, Valdemar Vicente Graciano. **Conceitos de arquiteturas de camadas**. Disponível em: <http://wiki.sintectus.com/bin/view/GrupoJava/ ConceitosDeArquiteturasDeCamadas>. Acesso em 10 nov. 2010.

CAETANO, Paulo. **Projeto arquitetural**: Disponível em: <https://disciplinas. dcc.ufba.br/pastas/MATA63/2009.1/Aula12-Projeto%20Arquitetural.pdf>. Acesso em: 20 nov. 2010.

SAUVÉ'S, Jacques Philippe. **Projeto de uma arquitetura**. Disponível em: < http://www.dsc.ufcg.edu.br/~jacques/cursos/apoo/html/proj1/proj2.htm#proj>. Acesso em: 15 nov. 2010.

SHALLOWAY, Alan; TROTT, James R. **Explicando padrões de projeto**: uma nova perspectiva em projeto orientado a objeto. São Paulo: Bookman, 2002.

TERRA, Ricardo; VALENTE, Marco Túlio. **Definição de padrões arquiteturais e seu impacto em atividades de manutenção de softwares.** VII Workshop de Manutenção de Software Moderna (WMSWM), p. 1-8, 2010. Disponível em: <http://homepages.dcc.ufmg.br/~mtov/pub.html>. Acesso em: 03 set. 2010.