CENTRO UNIVERSITÁRIO UNA

IMPLEMENTAÇÃO DE PADRÕES DE PROJETO
UTILIZANDO C#.NET

Marcos Paulo Marques Corrêa

Novembro de 2005

Marcos Paulo Marques Corrêa

IMPLEMENTAÇÃO DE PADRÕES DE PROJETO UTILIZANDO C#.NET

Orientador : Prof. Dirceu Belém

Trabalho de Conclusão do Curso Seqüencial de Formação Específica em Engenharia de Software do Centro Universitário UNA no segundo semestre de 2005.
Área de concentração: Desenvolvimento e Análise de Sistemas.

Belo Horizonte, novembro de 2005

Agradecimentos

A todos aqueles que, direta ou indiretamente, contribuíram para a realização deste trabalho, os meus agradecimentos sinceros e emocionados.

Em especial, agradeço:

Aos Professores Dirceu Belém e Mário Túlio, pelas orientações competentes, presentes e essenciais;

Aos Professores Mark Allan, Roberto Santos, Alexandre Ceolin, Rogério Rocha, Leonardo Prado, Uirá Endy, Alexandre Siqueira e Rogério Braga que ajudaram a despertar em mim o dom da programação;

Aos colegas Gleidison, Júlio, Samuel e Wanderci pela ajuda constante, troca de experiências e apoio incondicional;

Novamente ao colega Wanderci por abrir as portas do mercado de trabalho para mim.

Aos colegas Samuel, Marcelo, Leandro, Gleidison, João, Artur, Rangel, Vinícius e Alexandre por abrirem as portas de seus carros e me darem carona até algum lugar.

Á Tia Lena e à prima Tatiana por abrirem as portas de sua casa e me cederem um cantinho do seu lar para dormir e descansar durante esse último semestre.

Ao parceiro Cléber por acreditar no meu potencial e permitir que colocasse em prática na sua empresa tudo o que aprendi nesses anos.

Novamente ao Cléber e ao pessoal da Showmaker (Rafaela e Breno), por terem paciência e aceitarem que eu me dedicasse mais aos estudos do que à empresa.

E para finalizar, àquelas que são as pessoas mais importantes na minha vida, meu pai Paulo, minha mãe Suzana, minha irmã Tetê e minha namorada Júlia, por me apoiarem em minhas decisões e tanto me ajudarem em minha caminhada. Por tudo o que fizeram, fazem e farão por mim, amo vocês!

SUMÁRIO

1. INTRODUÇÃO 5
2. REFERENCIAL TEÓRICO 7
2.1. Programação Orientada a Objetos (POO) 7
2.2. Linguagem de Modelagem Unificada (UML) 9
2.2.1. Diagrama de Classes 10
2.3. Linguagem C#.NET 11
3. Padrões de Projeto 13
3.1. Criacionais 16
3.1.1. Construtor (Builder) 16
3.1.2. Fábrica (Factory) 19
3.1.3. Fábrica Abstrata (Abstract Factory) 21
3.1.4. Objeto Unitário (Singleton) 23
3.1.5. Protótipo (Prototype) 25
3.2. Estruturais 28
3.2.1. Adaptador (Adapter) 28
3.2.2. Composto (Composite) 31
3.2.3. Decorador (Decorator) 34
3.2.4. Fachada (Façade) 37
3.2.5. Peso Pena (Flyweight) 39
3.2.6. Ponte (Bridge) 42
3.2.7. Procurador (Proxy) 45
3.3. Comportamentais 48
3.3.1. Cadeia de Responsabilidade (Chain of Responsibility) 48
3.3.2. Comando (Command) 51
3.3.3. Estado (State) 54
3.3.4. Estratégia (Strategy) 57
3.3.5. Gabarito (Template) 60
3.3.6. Interpretador (Interpreter) 63
3.3.7. Iterador (Iterator) 66
3.3.8. Mediador (Mediator) 69
3.3.9. Observador (Observer) 72
3.3.10. Recordação (Memento) 74
3.3.11. Visitante (Visitor) 77
3.4. Freqüência de Uso 82
3.5. Relacionamento entre os Padrões 83
4. CONCLUSÃO 84
5. REFERÊNCIAS BIBLIOGRÁFICAS 85

ÍNDICE DE FIGURAS

Figura 1 – Exemplo de Diagrama de Classes 10
Figura 2 – Diagrama de Classes – Padrão Construtor 16
Figura 3 – Diagrama de Classes – Padrão Fábrica 19
Figura 4 – Diagrama de Classes – Padrão Fábrica Abstrata 21
Figura 5 – Diagrama de Classes – Padrão Objeto Unitário 24
Figura 6 – Diagrama de Classes – Padrão Protótipo 26
Figura 7 – Diagrama de Classes – Padrão Adaptador 28
Figura 8 – Diagrama de Classes – Padrão Composto 31
Figura 9 – Diagrama de Classes – Padrão Decorador 34
Figura 10 – Diagrama de Classes – Padrão Fachada 37
Figura 11 – Diagrama de Classes – Padrão Peso Pena 40
Figura 12 – Diagrama de Classes – Padrão Ponte 43
Figura 13 – Diagrama de Classes – Padrão Procurador 46
Figura 14 – Diagrama de Classes – Padrão Cadeia de Responsabilidade 48
Figura 15 – Diagrama de Classes – Padrão Comando 51
Figura 16 – Diagrama de Classes – Padrão Estado 54
Figura 17 – Diagrama de Classes – Padrão Estratégia 58
Figura 18 – Diagrama de Classes – Padrão Gabarito 60
Figura 19 – Diagrama de Classes – Padrão Interpretador 63
Figura 20 – Diagrama de Classes – Padrão Iterador 66
Figura 21 – Diagrama de Classes – Padrão Mediador 69
Figura 22 – Diagrama de Classes – Padrão Observador 72
Figura 23 – Diagrama de Classes – Padrão Recordação 75
Figura 24 – Diagrama de Classes – Padrão Visitante 78
Figura 25 - Gráfico de Frequência de Uso dos Padrões de Projeto 82
Figura 26 - Relacionamento entre Padrões de Projeto 83

RESUMO

Neste trabalho, objetivou-se analisar o catálogo de padrões de projeto apresentado pela Gang of Four (Gof), mostrando implementações e funcionalidades utilizando a linguagem de programação C#.NET. Através da análise orientada a objetos, juntamente com os diagramas de classes UML apresentados pelo grupo criador dos padrões, buscou-se uma implementação fiel aos modelos iniciais, porém voltados a casos de sistemas reais e possíveis, e não aos mesmos casos ilusórios e imaginários comumente mostrados por diversos autores. O referencial teórico foi organizado para atender às questões de pesquisa e incluiu autores e pesquisadores da ciência da computação. Os resultados foram estruturados em descrição da intenção do padrão, diagrama de classes genérico do mesmo, descrição e implementação de um projeto de sistema básico e simplista, porém real, onde o padrão pode ser utilizado. Ao final, são apresentados um gráfico e uma tabela contendo uma freqüência de utilização e o interrelacionamento entre os padrões, respectivamente.

1. INTRODUÇÃO

A busca pela reutilização de código sempre foi constante, desde o início da programação de computadores. Ao longo dos anos, os paradigmas estrutural, procedural, orientado a objetos (o qual é utilizado neste trabalho), e mais recentemente, orientado a aspectos, buscam cada vez mais uma maior reutilização de código, tornando assim as aplicações finais mais poderosas com um menor esforço do programador.

Porém a reutilização de código, bem como o projeto orientado a objetos, são tarefas difíceis de serem realizadas, sendo apenas conseguido com eficiência por projetistas experientes.

Surgiu daí então a idéia da criação de catálogos de padrões de projeto, os quais são uma espécie de manual de boas práticas a serem seguidas e utilizadas em projetos de software orientados a objetos. Dos catálogos mais conhecidos e utilizados, destaca-se o primeiro e considerado mais importante, escrito em 1995 pela Gang of Four, ou mais conhecida como GoF, e que também será o foco deste trabalho.

A intenção aqui é apresentar todos os 23 padrões de projeto contidos no catálogo exposto pela (GoF). Não é pretensão discorrer detalhadamente sobre todos os padrões, pois este trabalho já foi realizado no catálogo citado.

Apesar do catálogo GoF ser completo, detalhado e explicativo, ele possui o defeito de não possuir implementações reais, longe disso, ele apresenta exemplos de software ilusórios, pouco comuns e de pouca valia, além de serem apresentados em linguagens de difícil aprendizado como C++ e Smalltalk.

Este trabalho portanto tem o objetivo de apresentar os 23 padrões, suas intenções e objetivos, seus diagramas de classes e implementar um software simples, porém completo, onde o padrão pode ser utilizado.

A linguagem C#.NET foi escolhida para implementar os exemplos teste trabalho pois além de ser simples e completa, é uma linguagem moderna, completamente orientada a objetos, de fácil aprendizado e que pode ser utilizada em várias das diversas áreas que o desenvolvimento de softwares pode atingir atualmente, como a plataforma Windows, a plataforma Web e a plataforma de dispositivos móveis como celulares e smartphones.

2. REFERENCIAL TEÓRICO

2.1. Programação Orientada a Objetos (POO)

Antes do surgimento da programação Orientada a Objetos, os sistemas computacionais eram escritos utilizando linguagens procedurais. Tais linguagens permitiam dividir o código do sistema em módulos e submódulos, os quais continham funções que implementavam um algoritmo para realizar uma determinada tarefa.

Com o passar dos anos, percebeu-se que essa maneira de implementar os sistemas, através de módulos e algoritmos, não condizia com o mundo real. O mundo real é descrito através de *conceitos* como automóveis, casas e pessoas, e através de *entidades* concretas, como o meu carro palio, ou a sua casa amarela, ou o supervisor do funcionário João. Devido a essa incoerência, projetos e códigos tradicionais são pobremente mapeados para o mundo real.

Contudo, o paradigma de Orientação a Objetos veio para corrigir essa falha deixada pelos paradigmas anteriores. Braude (2005, p. 70) resume o principal objetivo da Orientação a Objetos como *fornecer um mapeamento direto entre o mundo real (conceitos) e as unidades de organização utilizadas no projeto (código)*.

As linguagens mais importantes que dão suporte à Orientação a Objetos são: Smalltalk, Perl, Python, PHP, C++, Java, C#.NET, VB.NET entre outras.

A seguir serão apresentados alguns conceitos básicos, relacionados à Orientação a Objeto.

- Classe: é um componente autônomo de um sistema. Um classe possui uma estrutura de uma abstração de dados do mundo real e as operações que podem ser realizadas sobre essa estrutura. O comportamento dos objetos (métodos) e os estados que ele é capaz de manter (atributos) são definidos pela classe.

- Objeto: é uma instância de uma classe, que realiza a classe para o sistema. Um objeto armazena estados através de seus atributos e reage a mensagens enviadas a ele; assim como se relaciona e envia mensagens a outros objetos.

- Mensagem: comunicação entre objetos que leva informação com a expectativa que resultará em uma atividade. O recebimento de uma mensagem é, normalmente, considerado um evento. (Deboni, p. 208)

- Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstratação de entidades existentes no domínio do sistema de software. (Wikipédia, http://pt.wikipedia.org/wiki/Abstra%C3%A7%C3%A3o)

- Encapsulamento: Segundo Braude (2005, p. 83) o termo *encapsulamento* é aplicado quando tentamos expor a funcionalidade, ocultado a maneira como essa funcionalidade é obtida (a "implementação").

- Herança é o mecanismo pelo qual uma "classe-pai" (super-classe) pode ser herdada por uma "classe-filho" (sub-classe), afim de aproveitar seus comportamentos (métodos) e seus estados possíveis (atributos).

- Herança Múltipla: ocorre quando uma sub-classe herda de mais de uma super-classe, possível apenas em algumas linguagens, como, por exemplo C++.

- Polimorfismo: permite um objeto se comportar de acordo com sua classe. Desta forma, é possível se relacionar com objetos de classes diferentes, enviar mensagens iguais (chamadas a métodos com mesma assinatura) e deixar o objeto se comportar à definição de sua classe. (Wikipédia, http://pt.wikipedia.org/wiki/Polimorfismo).

2.2. Linguagem de Modelagem Unificada (UML)

> *A UML (Unified Modeling Language ou Linguagem de Modelagem Unificada) é uma linguagem para especificar, visualizar, construir e documentar os artefatos de sistemas de software, bem como para modelar negócios e outros sistemas que não sejam de software* (OMG, 2001).

Guedes (2004, p.17) define a UML como uma linguagem visual utilizada pra modelar sistemas computacionais por meio do paradigma de Orientação a Objetos. A UML tornou-se, nos últimos anos, a linguagem padrão de modelagem de software adotada internacionalmente pela indústria de Engenharia de Software.

Segundo Deboni (2003, p.106) a notação da UML é uma notação gráfica, na qual a descrição de um software é feita com símbolos gráficos padronizados que se relacionam formando diagramas. Cada diagrama apresenta uma visão parcial do sistema de software, e a união dos diagramas deve representar um único sistema de software. Os diagramas são descritos por nome e termos padronizados, que compõem um glossário próprio de termos que também faz parte da UML.

Segue abaixo uma lista dos diagramas utilizados na UML:

- Diagrama de Casos de Uso
- Diagrama de Classes
- Diagrama de Objetos
- Diagrama de Estrutura Composta
- Diagrama de Seqüência
- Diagrama de Colaboração (Comunicação na UML 2)
- Diagrama de Gráfico de Estados (Máquina de Estados na UML 2)
- Diagrama de Atividades
- Diagrama de Componentes
- Diagrama de Implantação

- Diagrama de Pacotes
- Diagrama de Interação Geral
- Diagrama de Tempo

2.2.1. Diagrama de Classes

Por se o diagrama mais utilizado, o mais importante da UML e o único que será encontrado neste trabalho, o autor resolveu explicar apenas o diagrama de classes.

O Diagrama de Classes serve de apoio para a maioria dos outros diagramas, criando uma visão estática das definições das classes. Ele ilustra os atributos e métodos das classes, além de estabelecer como as classes se relacionam e as trocas de informações entre si.

Segue abaixo um exemplo desse diagrama.

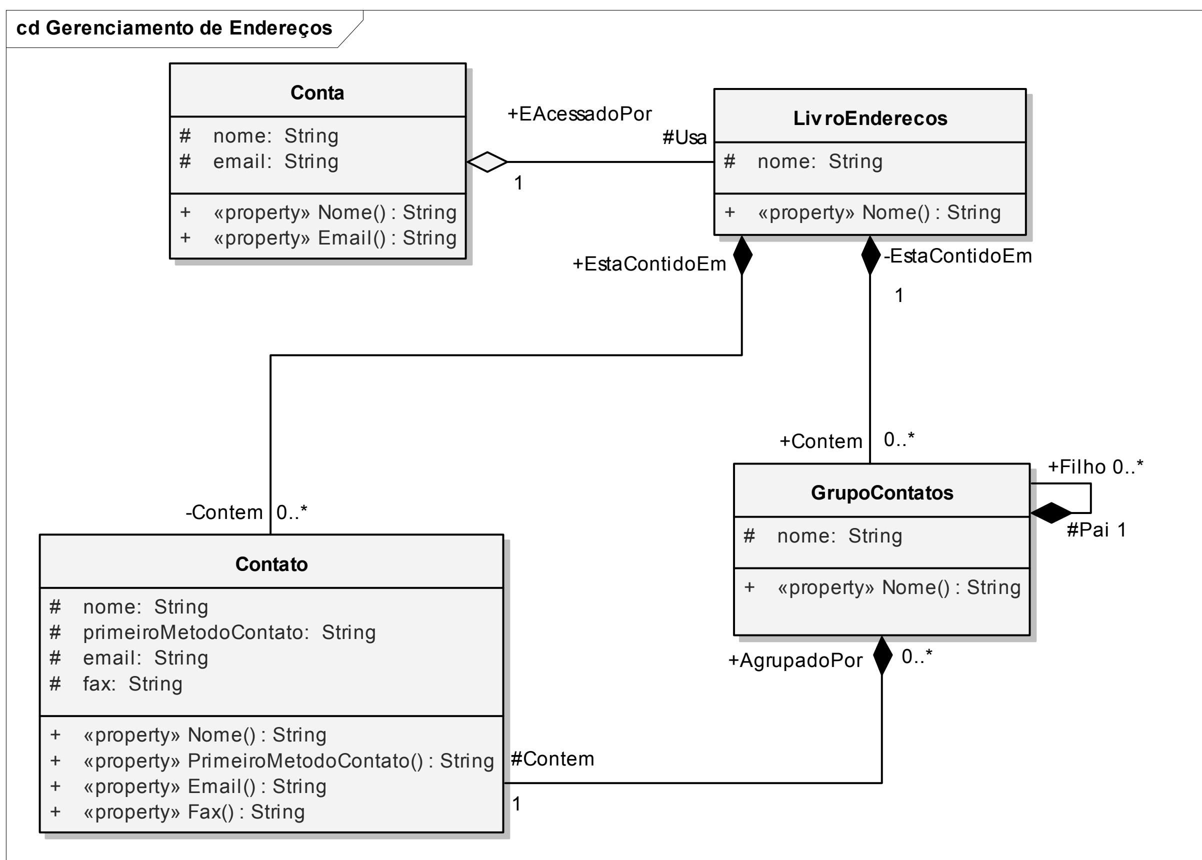

Figura 1 – Exemplo de Diagrama de Classes

2.3. Linguagem C#.NET

C# é uma linguagem de programação orientada a objetos baseada em C++ e JAVA. Foi desenvolvida pela Microsoft, mais especificamente por, Anders Hejlsberg, o mesmo criador do Turbo Pascal e do Delphi.

C# é uma linguagem bem simples, o que não a torna menos poderosa que outras linguagens. Com ela é possível escrever sofisticadas aplicações tanto para pequenas empresas quanto para grandes corporações.

Segue abaixo uma lista de algumas funcionalidades presentes na linguagem C#:

- Ponteiros e aritmética sem checagem só podem ser utilizados em uma modalidade especial chamada modo inseguro (*unsafe mode*). Normalmente os acessos a objetos é realizada através de referências seguras, as quais não podem ser invalidadas e normalmente as operações aritméticas são checadas contra sobrecarga (*overflow*).

- Objetos não são liberados explicitamente, mas através de um processo de coleta de lixo (*garbage collection*) quando não há referências aos mesmos, previnindo assim referências inválidas.

- Destrutores não existem. O equivalente mais próximo é a interface *Disposable*, que juntamente com a construção *using block* permitem que recursos alocados por um objeto sejam liberados prontamente. Também existem finalizadores, mas como em Java sua execução não é imediata.

- Como em Java, só é permitida a herança simples, mas uma classes pode implementar várias interfaces abstratas. O objetivo principal é simplificar a implementação do ambiente de execução.

- C# é mais seguro com tipos que C++. As únicas conversões implícitas por default são conversões seguras, tais como ampliação de inteiros e conversões de um tipo derivado para um tipo base. Não existem conversões implícitas entre inteiros e variáveis

lógicas ou enumerações. Não existem ponteiros nulos (*void pointers*) (apesar de referências para *Object* serem parecidas). E qualquer conversão implícita definida pelo usuário deve ser marcada explicitamente, diferentemente dos construtores de cópia de C++.

- A sintaxe para a declaração de vetores é diferente ("int[] a = new int[5]" ao invés de "int a[5]").
- C# não possui modelos (*templates*), mas C# 2.0 possui genéricos (*generics*).
- Propriedades estão disponíveis, as quais permitem que métodos sejam chamados com a mesma sintaxe de acesso a membros de dados.
- Recursos de reflexão completos estão disponíveis

3. Padrões de Projeto

As soluções orientadas a objetos, desde o seu início, sempre buscaram através dos mecanismos de herança uma maior reaproveitamento e reutilização de código.

Porém, se projetar código orientado a objetos já é difícil, mais difícil ainda é projetar software orientado a objetos com código reutilizável.

Projetistas experientes conseguem visualizar a reutilização facilmente na fase de projeto de uma aplicação, enquanto projetistas novatos têm mais dificuldade para conseguir este nível de abstração de dados a ponto de visualizar um padrão dentro do software.

Isso ocorre por que um projetista experiente, por ter trabalhado em vários projetos e visto vários sistemas diferentes, consegue encontrar padrões em códigos, de classes e de comunicação entre objetos.

Daí surgiu a idéia de padrões de projetos. Os padrões de projeto (do inglês *Design Patterns*) são soluções para problemas recorrentes no desenvolvimento de sistemas de software orientado a objetos.

Para Gamma (p. 18) o objeto primordial dos padrões de projeto é *capturar a experiência de projeto de uma forma que as pessoas possam usá-la efetivamente*.

Cristopher Alexander *apud* Gamma (p. 19) afirma: “cada padrão descreve um prolema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”.

Braude (p. 158) descreve os padrões de projeto como *combinações de classes e algoritmos associados que cumpre com propósitos comuns de* projeto. Para Braude um padrão expressa uma idéia em vez de uma combinação fixa de classes.

Com essa finalidade em vista, vários autores e bons projetistas escreveram e escrevem até hoje catálogos referente a padrões de projetos, alguns mais abrangente e outros mais específicos.

Um destes catálogos e considerado o pioneiro, mais importante, mais abrangente e mais utilizado, foi escrito em 1995 por Erich Gamma, Richar Helm, Ralph Johnson e John Vlissides, mais conhecidos como GoF (Gang of Four ou em português, Gangue dos Quatro), e se chama *Padrões de Projeto, soluções reutilizáveis de software orientado a objetos*, ou apenas *Design Patterns*. Larman (p. 352) descreve os padrões GoF, como são conhecidos, como *um trabalho embrionário e muito apreciado popular que apresenta 23 padrões úteis durante o projeto de objetos*.

Segue abaixo uma lista de características referente aos padrões de projeto, retirados do livro *Design Patterns:*

- tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas;
- técnicas testadas e aprovadasm se tornam mais acessível aos desenvolvedores de novos sistemas;
- ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização;
- melhoram a documentação e a manutenção de sistemaos ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacente.
- Ajudam um projetista a obter mais rapidamente um projeto adequado.

Deste ponto em diante entende-se como padrões ou padrões de projeto, os 23 padrões catalogados pelo GoF.

Os padrões variam na sua granularidade e no seu nível de abstração. Eles podem ser classificados em dois critérios, o primeiro deles é quanto à finalidade, ou seja, o que um padrão faz. As 3 finalidades possíveis são:

Criação – se preocupam com o processo de criação de objetos;

Estruturais – lidam com a composição de classes ou de objetos;

Comportamentais – caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades;

O segundo critério de classificação é quanto ao escopo, ele especifica se o padrão se aplica primariamente a classes ou a objetos.

Segue abaixo uma tabela retirada do catálogo GoF, que nomeia os padrões e os classifica tanto ao propósito quanto ao escopo:

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="3">Propósito</td></tr>
<tr><td>Criacional</td><td>Estrutural</td><td>Comportamental</td></tr>
<tr><td rowspan="2">Escopo</td><td>Classe</td><td>Fábrica</td><td>Adaptador (classe)</td><td>Interpretador
Gabarito</td></tr>
<tr><td>Objeto</td><td>Fábrica Abstrata
Construtor
Protótipo
Objeto Unitário</td><td>Adaptador (objeto)
Ponte
Composto
Decorador
Fachada
Peso Pena
Procurador</td><td>Cadeia de Responsabilidade
Commando
Iterador
Mediador
Recordação
Observador
Estado
Estratégia
Visitante</td></tr>
</table>

Tabela 1 – Classificação de Padrões de Projeto GoF

Este trabalho não tem como objetivo explicar detalhadamente cada um dos 23 padrões GoF, até por que todos já estão detalhados no catálogo dos padrões, porém, serão mostrados exemplos de implementação de cada um dos padrões utilizando a linguagem C#, afim de demonstrar como eles podem ser utilizados com uma linguagem de última geração.

Para Gamma (p. 18), *cada padrão de projeto sistematicamente nomeia, explica e avalia um aspecto de projeto importante e recorrente em sistemas orientados a objetos.*

Este trabalho, como é voltado mais para implementação, irá apresentar a intenção do projeto segundo um ou dois autores, um diagrama de classe conceitual do padrão apresentado e uma implementação de um pequeno programa utilizando a linguagem C#.

3.1. Criacionais

3.1.1. Construtor (Builder)

Intenção

Para Gamma (p. 104) a intenção do padrão Construtor é separar a construção de um objeto complexo, da sua representação. Com isso, consegue-se criar diferentes representações desse objeto complexo, utilizando o mesmo processo de construção.

Diagrama

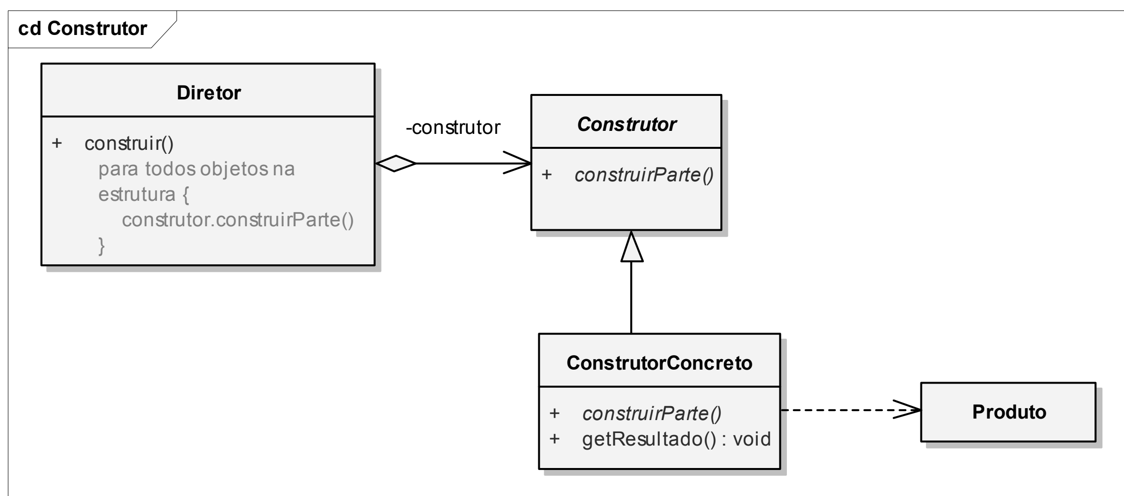

Figura 2 – Diagrama de Classes – Padrão Construtor

Explicação

O exemplo abaixo demonstra o padrão Construtor onde, diferentes veículos são criados em um modo passo-a-passo. A Montadora usa ConstrutorVeiculo para construir uma variedade de Veículo em uma série de passos seqüenciais.

Com isso, é possível termos vários Veículos com propriedades diferentes, utilizando o mesmo método de construção para cada um.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Criacional.Construtor
{
	public class Principal
	{
		public static void Main()
		{
			// Cria montadora com construtores de veículos
			Montadora montadora = new Montadora();
			ConstrutorVeiculo c1 = new ConstrutorMoto();
			ConstrutorVeiculo c2 = new ConstrutorCarro();

			// Construir and display veiculos
			montadora.Construir(c1);
			c1.Veiculo.Mostra();

			montadora.Construir(c2);
			c2.Veiculo.Mostra();
		}
	}

	// "Diretor"
	class Montadora
	{
		// Construtor usa uma série complexa de passos
		public void Construir(ConstrutorVeiculo construtorVeiculo)
		{
			construtorVeiculo.ConstruirChassi();
			construtorVeiculo.ConstruirMotor();
			construtorVeiculo.ConstruirRodas();
			construtorVeiculo.ConstruirPortas();
		}
	}

	// "Construtor"
	abstract class ConstrutorVeiculo
	{
		protected Veiculo veiculo;

		// Propriedade
		public Veiculo Veiculo
		{
			get{ return veiculo; }
		}

		public abstract void ConstruirChassi();
		public abstract void ConstruirMotor();
		public abstract void ConstruirRodas();
		public abstract void ConstruirPortas();
	}

	// "ConstrutorConcreto1"
	class ConstrutorMoto : ConstrutorVeiculo
	{
		public override void ConstruirChassi()
		{
```

```
                veiculo = new Veiculo("Moto");
                veiculo["chassi"] = "Chassi da Moto";
        }

        public override void ConstruirMotor()
        {
                veiculo["motor"] = "500 cc";
        }

        public override void ConstruirRodas()
        {
                veiculo["rodas"] = "2";
        }

        public override void ConstruirPortas()
        {
                veiculo["portas"] = "0";
        }
}

// "ConstrutorConcreto2"
class ConstrutorCarro : ConstrutorVeiculo
{
        public override void ConstruirChassi()
        {
                veiculo = new Veiculo("Carro");
                veiculo["chassi"] = "Chassi do Carro";
        }

        public override void ConstruirMotor()
        {
                veiculo["motor"] = "2500 cc";
        }

        public override void ConstruirRodas()
        {
                veiculo["rodas"] = "4";
        }

        public override void ConstruirPortas()
        {
                veiculo["portas"] = "4";
        }
}

// "Produto"
class Veiculo
{
        private string tipo;
        private Hashtable partes = new Hashtable();

        // Construtor (da classe)
        public Veiculo(string tipo)
        {
                this.tipo = tipo;
        }

        // Indexador
        public object this[string chave]
        {
                get{ return partes[chave]; }
                set{ partes[chave] = value; }
        }

        public void Mostra()
        {
                Console.WriteLine("\n---------------------------");
                Console.WriteLine("Tipo do Veículo: {0}", tipo);
```

```
                    Console.WriteLine(" Chassi : {0}", partes["chassi"]);
                    Console.WriteLine(" Motor : {0}", partes["motor"]);
                    Console.WriteLine(" #Rodas: {0}", partes["rodas"]);
                    Console.WriteLine(" #Portas : {0}", partes["portas"]);
                }
            }
        }
```

3.1.2. Fábrica (Factory)

Intenção

Define uma interface para criar um objeto, mas deixa as subclasses decidirem que classe instanciar (Gamma, p. 104).

Diagrama

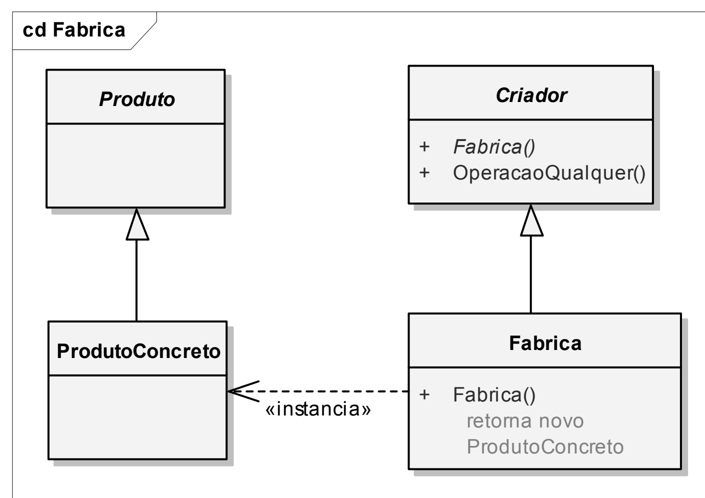

Figura 3 – Diagrama de Classes – Padrão Fábrica

Explicações

O código abaixo mostra a flexibilidade oferecida pela Fábrica na criação de diferentes fabricantes de carros. As classes Volkswagen e Fiat, derivadas de Fabricante, instanciam versões extendidas da classe Fabricante. No exemplo, a Fábrica é chamada no construtor da classe base Fabricante.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Criacional.Fabrica
{
```

```
// Aplicação de Teste Principal
class Principal
{
        static void Main()
        {
                // Nota: construtores chamam a Fábrica
                Fabricante[] fabricantes = new Fabricante[2];
                fabricantes[0] = new Volkswagen();
                fabricantes[1] = new Fiat();

                // Mostra carros dos Fabricantes
                foreach (Fabricante fabricante in fabricantes)
                {
                        Console.WriteLine("\n" + fabricante.GetType().Name+
"--");
                        foreach (Carro carro in fabricante.Carros)
                        {
                                Console.WriteLine(" " +
carro.GetType().Name);
                        }
                }
        }
}

// "Produto"
abstract class Carro
{
}

// "ProdutosConcreto"
class Golf : Carro
{
}

class Polo : Carro
{
}

class Marea : Carro
{
}

class Idea : Carro
{
}

// "Criador"
abstract class Fabricante
{
        private ArrayList carros = new ArrayList();

        // Construtor chama a Fábrica
        public Fabricante()
        {
                this.CriaCarros();
        }

        public ArrayList Carros
        {
                get{ return carros; }
        }

        // Fábrica
        public abstract void CriaCarros();
}

// "Criadores Concretos"
```

```
        class Volkswagen : Fabricante
        {
                // Implementação da Fábrica
                public override void CriaCarros()
                {
                        Carros.Add(new Golf());
                        Carros.Add(new Polo());
                }
        }

        class Fiat : Fabricante
        {
                // Implementação da Fábrica
                public override void CriaCarros()
                {
                        Carros.Add(new Marea());
                        Carros.Add(new Idea());
                }
        }
}
```

3.1.3. Fábrica Abstrata (Abstract Factory)

Intenção

Fornece uma interface que facilita a criação de famílias de objetos que tenham algum relacionamento ou que sejam dependentes, sem a necessidade de especificar suas classes concretas (Gamma, p. 95).

Diagrama

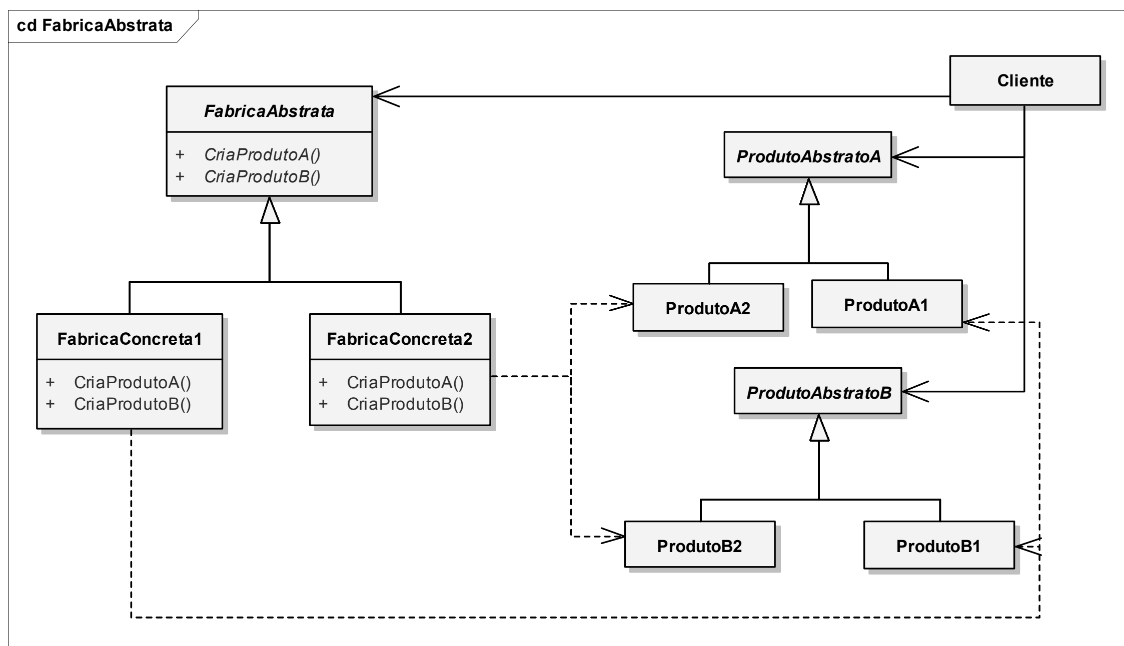

Figura 4 – Diagrama de Classes – Padrão Fábrica Abstrata

Explicações

O código abaixo mostra a criação de diferentes montadoras de veículos para rodar, por exemplo, em uma aplicação de concessionária de veículos novos e semi-novos.

As fábricas concretas do diagrama são representadas pelas montadoras (Suzuki e Honda).

Ainda que as montadoras criem Produtos diferentes (Titan, Fit, Katana e Vitara), as interações entre as montadoras e os produtos são os mesmos, pois eles são definidos pelos tipos (ProdutosAbstratos A e B - Carros e Motos).

Código

```
using System;

namespace TCC.MP.Criacional.FabricaAbstrata
{
	class Principal
	{
		public static void Main()
		{
			// Cria a FabricaHonda com seus respectivos automóveis
			FabricaHonda honda = new FabricaHonda();
			Automoveis auto = new Automoveis(honda);


			// Cria a FabricaSuzuki com seus respectivos automóveis
			FabricaSuzuki suzuki = new FabricaSuzuki();
			auto = new Automoveis(suzuki);
		}
	}

	// "FabricaAbstrata"
	abstract class FabricaMontadora
	{
		public abstract Moto CriaMoto();
		public abstract Carro CriaCarro();
	}

	// "FabricaConcreta1"
	class FabricaHonda : FabricaMontadora
	{
		public override Moto CriaMoto()
		{
			return new Titan();
		}
		public override Carro CriaCarro()
		{
			return new Fit();
		}
	}

	// "FabricaConcreta2"
	class FabricaSuzuki : FabricaMontadora
	{
```

```
                public override Moto CriaMoto()
                {
                        return new Katana();
                }
                public override Carro CriaCarro()
                {
                        return new Vitara();
                }
        }

        // "ProdutoAbstratoA"
        abstract class Moto
        {
        }

        // "ProdutoAbstratoB"
        abstract class Carro
        {
        }

        // "ProdutoA1"
        class Titan : Moto
        {
        }

        // "ProdutoB1"
        class Fit : Carro
        {
        }

        // "ProdutoA2"
        class Katana : Moto
        {
        }

        // "ProdutoB2"
        class Vitara : Carro
        {
        }

        // "Cliente"
        class Automoveis
        {
                private Moto moto;
                private Carro carro;

                public Automoveis(FabricaMontadora fabrica)
                {
                        moto = fabrica.CriaMoto();
                        carro = fabrica.CriaCarro();
                }
        }
}
```

3.1.4. Objeto Unitário (Singleton)

Intenção

Garante que exista apenas uma instância de uma classe e fornece um ponto global para acesso a essa instância (Gamma, p.130).

Diagrama

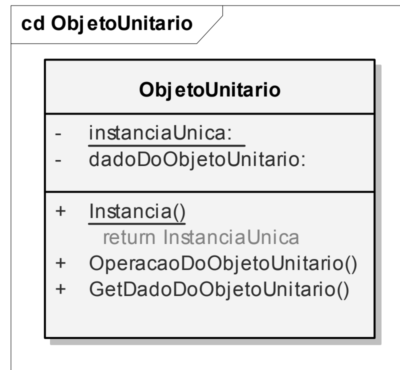

Figura 5 – Diagrama de Classes – Padrão Objeto Unitário

Explicações

O código abaixo demonstra a funcionalidade do padrão Objeto Unitário. A Construtora da classe Logging é definida como privada, prevenindo que a classe não seja instanciada. O instanciamento da classe será realizado através do método Instancia, que verifica se o objeto "instancia" já foi instanciado. Caso ainda não tenha sido, o método se encarrega de executar a operação e assim retorna o objeto unitário.

Com isso, a instância da classe Logging pode ser chamada em qualquer lugar dentro da aplicação e ainda assim a classe garante que apenas uma instância será criada.

Código

```
using System;
using System.Collections;

namespace Monografia.MarcosPaulo.ObjetoUnitario
{
	class Principal
	{
		static void Main(string[] args)
		{
			Logging log1 = Logging.Instancia;
			log1.registros.Add("Instancia 1 de Logging!");

			Logging log2 = Logging.Instancia;
			log2.registros.Add("Instancia 2 de Logging!");

			Logging log3 = Logging.Instancia;
			log3.mostraRegistros();
		}
	}

	// Classe Objeto Unitário
	public class Logging
	{
		// instancia estática do próprio Objeto
```

```
            static Logging instancia = null;
            // outro atributo qualquer da classe
            public ArrayList registros = new ArrayList();

            // construtor privado previne a
            // não instanciação da classe
            private Logging()
            {
            }

            public static Logging Instancia
            {
                  get
                  {
                        //instancia a classe caso nao tenha sido
                        if(instancia == null)
                        {
                              instancia = new Logging();
                        }
                        //retorna o objeto unitário
                        return instancia;

                  }
            }

            // outro método qualquer da classe
            public void mostraRegistros()
            {
                  Console.WriteLine(" Registros Gravados\n");
                  foreach (String registro in registros)
                  {
                        Console.WriteLine("  " + registro + "\n");
                  }
            }
      }
}
```

3.1.5. Protótipo (Prototype)

Intenção

Criar um conjunto de objetos quase idênticos, clonando-os. O tipo desses objetos é determinado em tempo de execução (Braude, p.200)

Diagrama

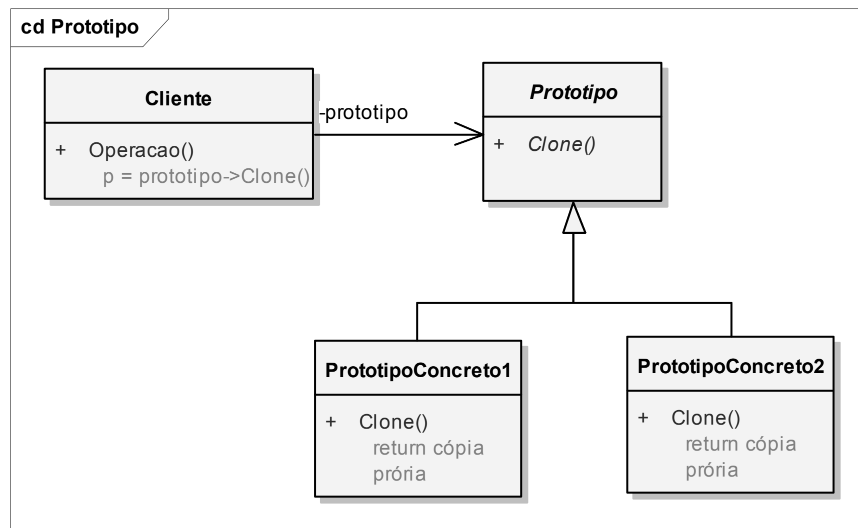

Figura 6 – Diagrama de Classes – Padrão Protótipo

Explicação

O exemplo abaixo mostra uma implementação bem específica do padrão Protótipo, onde um "GerenciadorDeCarros", que no caso poderia ser uma concessionária de carros novos e semi-novos, é inicializado com valores de carros padrões (Idea, Golf, Astra).

Posteriormente obtemos cópias idênticas destes carros utilizando o método Clone().

Caso existisse uma maneira de alterar as propriedades dos carros (marca, modelo, ano, cor e preço), seria possível obter carros ligeiramente parecidos.

Como este não é o caso do exemplo abaixo, apenas é possível obter cópias idênticas dos carros padrões.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Criacional.Prototipo
{
	class Principal
	{
		static void Main()
		{
			GerenciadorDeCarro gerenciador = new GerenciadorDeCarro();

			// Inicializa com carros padrões
			gerenciador["idea"] = new Carro("Fiat", "Idea", "laranja",
2006, 40000.00);
			gerenciador["golf"] = new Carro("VW", "Golf", "preto",
```

```
2005, 35000.00);
			gerenciador["astra"] = new Carro("Chevrolet", "Astra",
"prata", 2003, 30000.00);

			Carro carro;

			// Usuario usa carros padrões
			string nome = "idea";
			carro = gerenciador[nome].Clone() as Carro;

			nome = "golf";
			carro = gerenciador[nome].Clone() as Carro;

			nome = "astra";
			carro = gerenciador[nome].Clone() as Carro;
		}
	}

	// "Prototipo"
	abstract class CarroPrototipo
	{
		public abstract CarroPrototipo Clone();
	}

	// "PrototipoConcreto"
	class Carro : CarroPrototipo
	{
		private string marca;
		private string modelo;
		private string cor;
		private int ano;
		private double preco;

		// Construtor
		public Carro(string marca, string modelo, string cor, int ano,
double preco)
		{
			this.marca = marca;
			this.modelo = modelo;
			this.cor = cor;
			this.ano = ano;
			this.preco = preco;
		}

		// Cria uma cópia superficial do Carro
		public override CarroPrototipo Clone()
		{
			Console.WriteLine("Clonando carro: {0}, {1}, {2}, {3},
{4}", marca, modelo, cor, ano, preco);

			return this.MemberwiseClone() as CarroPrototipo;
		}
	}

	// Gerenciador de Protótipos
	class GerenciadorDeCarro
	{
		Hashtable carros = new Hashtable();

		// Indexador
		public CarroPrototipo this[string nome]
		{
			get
			{
				return carros[nome] as CarroPrototipo;
			}
			set
			{
```

```
                              carros.Add(nome, value);
                        }
                  }
            }
}
```

3.2.Estruturais

3.2.1. Adaptador (Adapter)

Intenção

Converte a interface de uma classe em outra interface esperada pelos clientes. Adaptador permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces. (Gamma, p.140)

Diagrama

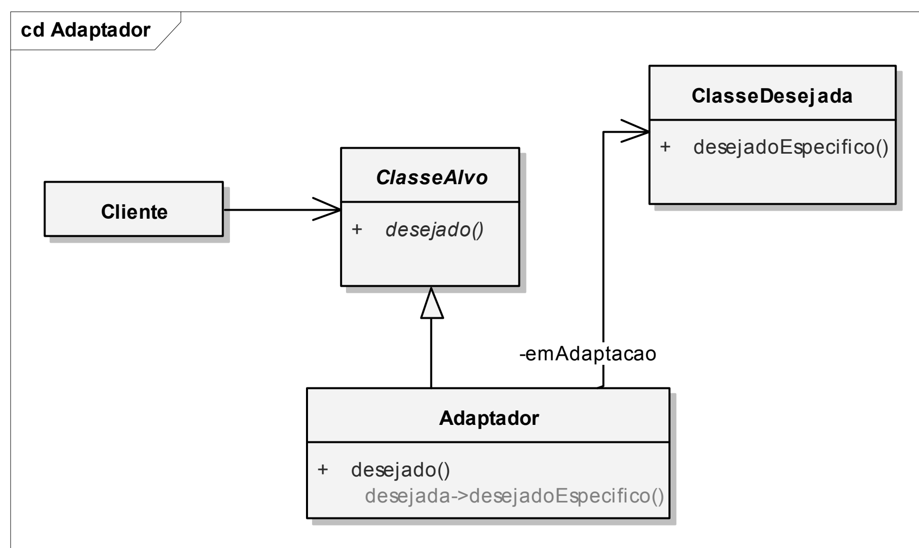

Figura 7 – Diagrama de Classes – Padrão Adaptador

Explicações

O exemplo abaixo demonstra o uso de uma base de dados químico legado. Os objetos de Compostos Químicos acessam a base de dados através de uma interface Adaptador.

Código

```
using System;

namespace TCC.MP.Estrutura.Adaptador
{
      class Principal
      {
```

```
        static void Main()
        {
                // Composto Químico não-adaptado
                Composto material = new Composto("Desconhecido");
                material.mostra();

                // Composto Químico adaptado
                Composto agua = new CompostoRico("Água");
                agua.mostra();

                Composto benzina = new CompostoRico("Benzina");
                benzina.mostra();

                Composto alcool = new CompostoRico("Álcool");
                alcool.mostra();
        }
}

// "ClasseAlvo"
class Composto
{
        protected string nome;
        protected float pontoEbulicao;
        protected float pontoFusao;
        protected double pesoMolecular;
        protected string formulaMolecular;

        // Construtor
        public Composto(string nome)
        {
                this.nome = nome;
        }

        public virtual void mostra()
        {
                Console.WriteLine("\nComposto: {0} ------ ", nome);
        }
}

// "Adaptador"
class CompostoRico : Composto
{
        private BaseDeDadosQuimica banco;

        // Construtor
        public CompostoRico(string nome) : base(nome)
        {
        }

        public override void mostra()
        {
                // Desejada
                banco = new BaseDeDadosQuimica();
                pontoEbulicao = banco.getPontoCritico(nome, "E");
                pontoFusao = banco.getPontoCritico(nome, "F");
                pesoMolecular = banco.getPesoMolecular(nome);
                formulaMolecular = banco.getEstruturaMolecular(nome);

                base.mostra();
                Console.WriteLine(" Fórmula: {0}", formulaMolecular);
                Console.WriteLine(" Peso : {0}", pesoMolecular);
                Console.WriteLine(" Pt. Ebuliçãot: {0}", pontoFusao);
                Console.WriteLine(" Pt. Fusão: {0}", pontoEbulicao);
        }
}

// "ClasseDesejada"
class BaseDeDadosQuimica
```

```
	{
		// Base de Dados da 'API legada'
		public float getPontoCritico(string composto, string point)
		{
			float temperatura = 0.0F;

			// Ponto de Ebulição
			if (point == "E")
			{
				switch (composto.ToLower())
				{
					case "água" : temperatura = 0.0F; break;
					case "benzina" : temperatura = 5.5F; break;
					case "álcool" : temperatura = -114.1F; break;
				}
			}
			// Ponto de Fusão
			else
			{
				switch (composto.ToLower())
				{
					case "água" : temperatura = 100.0F; break;
					case "benzina" : temperatura = 80.1F; break;
					case "álcool" : temperatura = 78.3F; break;
				}
			}
			return temperatura;
		}

		public string getEstruturaMolecular(string composto)
		{
			string estrutura = "";

			switch (composto.ToLower())
			{
				case "água" : estrutura = "H20"; break;
				case "benzina" : estrutura = "C6H6"; break;
				case "álcool" : estrutura = "C2H6O2"; break;
			}
			return estrutura;
		}

		public double getPesoMolecular(string composto)
		{
			double peso = 0.0;
			switch (composto.ToLower())
			{
				case "água" : peso = 18.015; break;
				case "benzina" : peso = 78.1134; break;
				case "álcool" : peso = 46.0688; break;
			}
			return peso;
		}
	}
}
```

3.2.2. Composto (Composite)

Intenção

Compor objetos em estruturas de árvore para representarem hierarquias partes-todo. (Gamma, p.160)

Diagrama

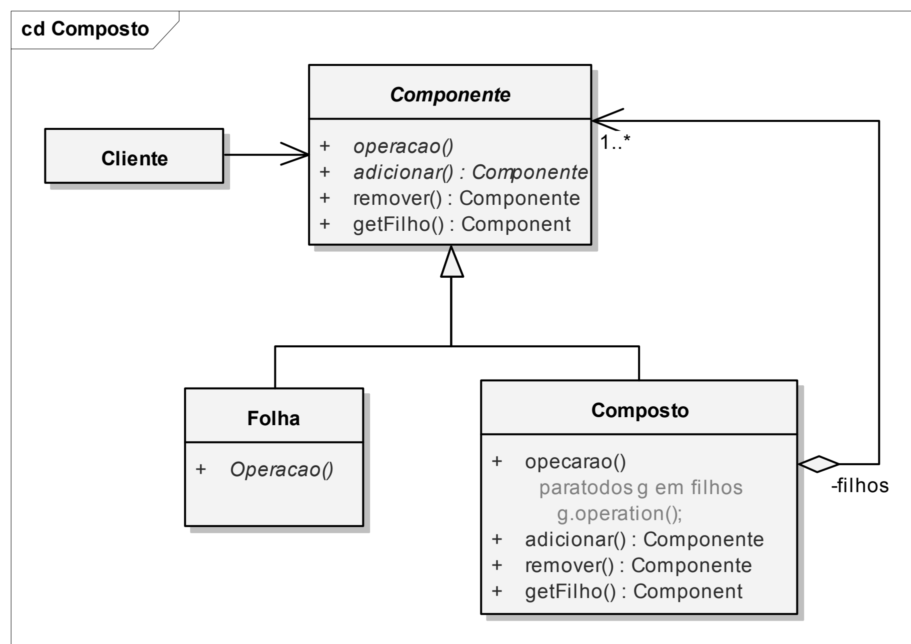

Figura 8 – Diagrama de Classes – Padrão Composto

Explicações

O código de exemplo abaixo demonstra a funcionalidade do padrão Composto na construção de estruturas de árvore onde temos nós primitivos (ElementoPrimitivo: Fuselagem, Cauda, Nariz, Motores e Tanques) compondo nós compostos (ElementoComposto, no caso um avião Airbus A-319).

As classes nós (Elemento, ElementoPrimitivo e ElementoComposto) abaixo, podem ser utilizadas em qualquer sistema que utilize a estrutura de árvores, só é necessária a mudança no programa principal, que irá fazer a composição dos elementos, dependendo da necessidade da aplicação.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Estrutural.Composto
{
	class Principal
	{
		static void Main()
		{
			// Cria uma estrutura de árvore
			ElementoComposto aviao = new ElementoComposto("Airbus A-
319");
			aviao.adiciona(new ElementoPrimitivo("Fuselagem"));
			aviao.adiciona(new ElementoPrimitivo("Cauda"));
			aviao.adiciona(new ElementoPrimitivo("Nariz"));


			// Cria um nó não folha
			ElementoComposto asaDireita = new ElementoComposto("Asa
Direita");
			asaDireita.adiciona(new ElementoPrimitivo("Motor
Direito"));
			asaDireita.adiciona(new ElementoPrimitivo("Tanque
Direito"));
			aviao.adiciona(asaDireita);

			// Cria outro nó não folha
			ElementoComposto asaEsquerda = new ElementoComposto("Asa
Esquerda");
			asaEsquerda.adiciona(new ElementoPrimitivo("Motor
Esquerdo"));
			asaEsquerda.adiciona(new ElementoPrimitivo("Tanque
Esquerdo"));
			aviao.adiciona(asaEsquerda);

			// adiciona e remove um ElementoPrimitivo
			ElementoPrimitivo tremDePouso = new
ElementoPrimitivo("Trem de Pouso");
			aviao.adiciona(tremDePouso);
			aviao.remove(tremDePouso);

			// mostra os nós recursivamente
			aviao.mostra(1);
		}
	}

	// "Componente" Nó da Árvore
	abstract class Elemento
	{
		protected string nome;

		// Construtor
		public Elemento(string nome)
		{
			this.nome = nome;
		}

		public abstract void adiciona(Elemento d);
		public abstract void remove(Elemento d);
		public abstract void mostra(int indentacao);
	}

	// "Folha"
	class ElementoPrimitivo : Elemento
	{
```

```
			// Construtor
			public ElementoPrimitivo(string nome) : base(nome)
			{
			}

			public override void adiciona(Elemento c)
			{
				Console.WriteLine("Não é possível adicionar a 
ElementoPrimitivo");
			}

			public override void remove(Elemento c)
			{
				Console.WriteLine("Não é possível remover de 
ElementoPrimitivo");
			}

			public override void mostra(int indentacao)
			{
				Console.WriteLine(new String('-', indentacao) + " " + 
nome);
			}
		}

		// "Composto"
		class ElementoComposto : Elemento
		{
			private ArrayList elementos = new ArrayList();

			// Construtor
			public ElementoComposto(string nome) : base(nome)
			{
			}

			public override void adiciona(Elemento d)
			{
				elementos.Add(d);
			}

			public override void remove(Elemento d)
			{
				elementos.Remove(d);
			}

			public override void mostra(int indentacao)
			{
				Console.WriteLine(new String('-', indentacao) + "+ " + 
nome);

				// mostra cada elemento filho neste nó
				foreach (Elemento e in elementos)
					e.mostra(indentacao + 2);
			}
		}
}
```

3.2.3. Decorador (Decorator)

Intenção

Agregar responsabilidades adicionais dinamicamente a um objeto, fornecendo assim, uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.

Diagrama

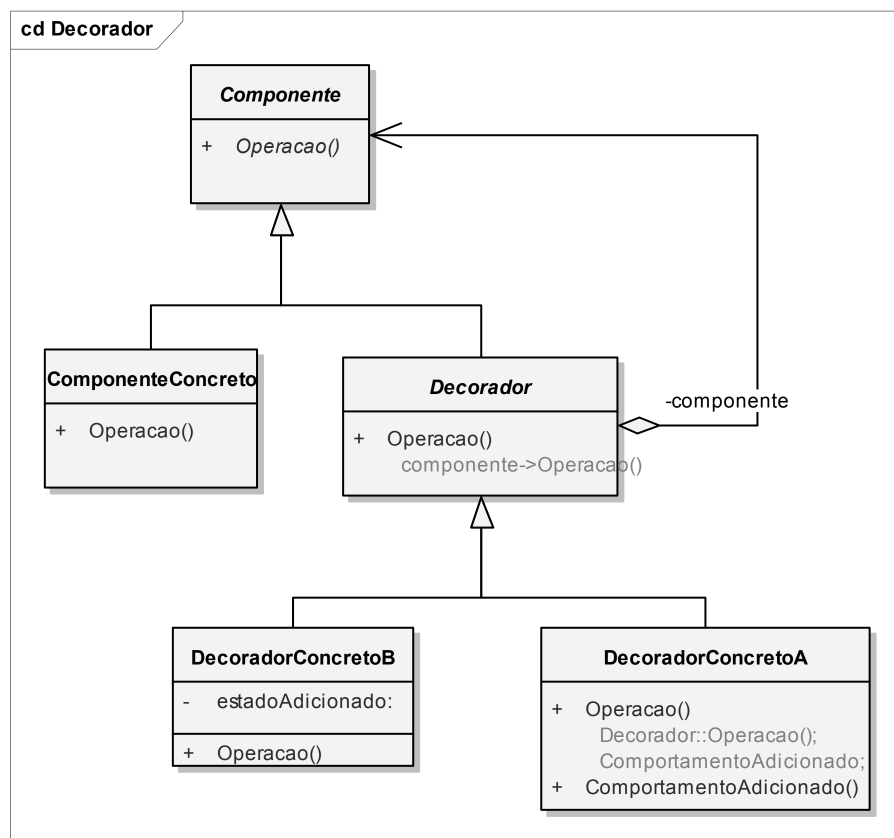

Figura 9 – Diagrama de Classes – Padrão Decorador

Explicações

O exemplo de código a seguir mostra uma funcionalidade do padrão Decorador, onde a funcionalidade Emprestável é adicionada a um item da biblioteca existente (livro ou vídeo).

Código

```
using System;
using System.Collections;

namespace TCC.MP.Estrutural.Decorador
{
	class Principal
	{
		static void Main()
		{
```

```
                    // Cria Livro
                    Livro livro = new Livro ("E. Gamma", "Padrões de Projeto",
10);
                    livro.Mostra();

                    // Cria Video
                    Video video = new Video ("B. Gates", "Eu adoro C#", 23,
92);
                    video.Mostra();

                    // Faz o vídeo emprestável, depois empresta e mostra
                    Console.WriteLine("\nEmprestando Vídeos:");

                    Emprestavel emprestarVideo = new Emprestavel(video);
                    emprestarVideo.EmprestarItem("Cliente nº 1");
                    emprestarVideo.EmprestarItem("Cliente nº 2");
                    emprestarVideo.Mostra();
             }
       }

       // "Componente"
       abstract class ItemDaBiblioteca
       {
             private int numCopias;

             // Propriedade
             public int NumCopias
             {
                    get{ return numCopias; }
                    set{ numCopias = value; }
             }

             public abstract void Mostra();
       }

       // "ComponenteConcreto"
       class Livro : ItemDaBiblioteca
       {
             private string autor;
             private string titulo;

             // Construtor
             public Livro(string autor,string titulo,int numCopias)
             {
                    this.autor     = autor;
                    this.titulo    = titulo;
                    this.NumCopias = numCopias;
             }

             public override void Mostra()
             {
                    Console.WriteLine("\nLivro ------ ");
                    Console.WriteLine(" Autor: {0}", autor);
                    Console.WriteLine(" Título: {0}", titulo);
                    Console.WriteLine(" Nº Cópias: {0}", NumCopias);
             }
       }

       // "ComponenteConcreto"
       class Video : ItemDaBiblioteca
       {
             private string diretor;
             private string titulo;
             private int duracao;

             // Construtor
             public Video(string diretor, string titulo,
                    int numCopias, int duracao)
```

```
		{
			this.diretor = diretor;
			this.titulo = titulo;
			this.NumCopias = numCopias;
			this.duracao = duracao;
		}

		public override void Mostra()
		{
			Console.WriteLine("\nVídeo ----- ");
			Console.WriteLine(" Diretor: {0}", diretor);
			Console.WriteLine(" Title: {0}", titulo);
			Console.WriteLine(" Duração: {0}\n", duracao);
			Console.WriteLine(" # Copies: {0}", NumCopias);
		}
	}

	// "Decorador"
	abstract class Decorator : ItemDaBiblioteca
	{
		protected ItemDaBiblioteca itemDaBiblioteca;

		// Construtor
		public Decorator(ItemDaBiblioteca itemDaBiblioteca)
		{
			this.itemDaBiblioteca = itemDaBiblioteca;
		}

		public override void Mostra()
		{
			itemDaBiblioteca.Mostra();
		}
	}

	// "DecoradorConcreto"
	class Emprestavel : Decorator
	{
		protected ArrayList clientes = new ArrayList();

		// Construtor
		public Emprestavel(ItemDaBiblioteca itemDaBiblioteca)
			: base(itemDaBiblioteca)
		{
		}

		public void EmprestarItem(string nome)
		{
			clientes.Add(nome);
			itemDaBiblioteca.NumCopias--;
		}

		public void DevolverItem(string nome)
		{
			clientes.Remove(nome);
			itemDaBiblioteca.NumCopias++;
		}

		public override void Mostra()
		{
			base.Mostra();

			foreach (string cliente in clientes)
			{
				Console.WriteLine(" Cliente: " + cliente);
			}
		}
	}
}
```

3.2.4. Fachada (Façade)

Intenção

Fornece uma interface unificada para o acesso a um conjunto de interfaces em um subsistema. (Gamma, p.179)

Diagrama

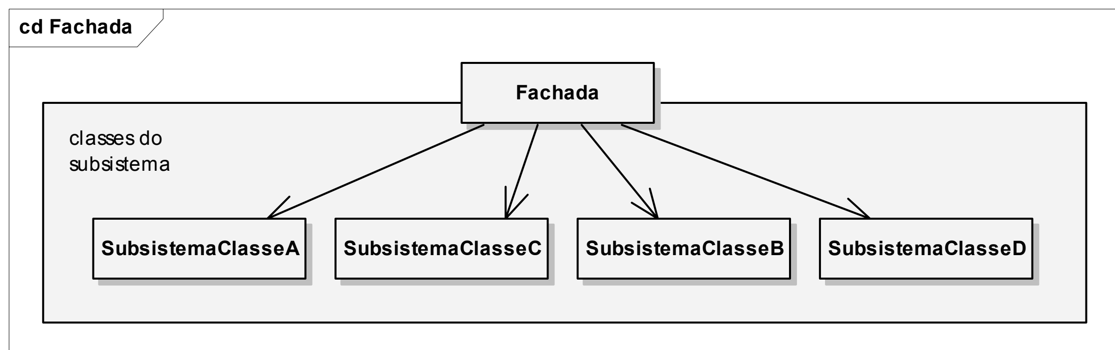

Figura 10 – Diagrama de Classes – Padrão Fachada

Explicações

No exemplo abaixo o padrão Fachada foi utilizado na classe Contratação. Essa classe provê uma interface simplificada para um sistema complexo de avaliação de contratação de candidatos de uma empresa.

No exemplo foram utilizados 4 subsistemas (ExameMedico, ExamePsicotecnico, ProvaPratica e PretensaoSalarial) para avaliar o perfil do candidato. Caso ele seja aprovado nos 4 quesitos, será contratado.

No caso apresentado, todos os subsistemas (com exceção da PrentesaoSalarial) retornarão verdadeiro, ou seja, o candidato sempre será contratado. Em uma aplicação real, poderiam e deveriam ser utilizadas consultas a algumas bases de dados, e/ou chamadas a outras classes para a devida validação.

Código

```
using System;

namespace TCC.MP.Estrutural.Fachada
{
	class Principal
```

```
	{
		static void Main(string[] args)
		{
			//Fachada
			Contratacao contratacao = new Contratacao();

			//Avalia a possibilidade de Contratacao do Candidato
			Candidato candidato = new Candidato("Marcos Paulo",
10000);
			bool contratar = contratacao.seraContratado(candidato);

			Console.WriteLine("\n"+ candidato.Nome + (contratar ? "" :
"Não") + " será contratado!");
		}
	}

	//SubsistemaClasseA
	class ExameMedico
	{
		public bool possuiCondicoesFisicas(Candidato c)
		{
			Console.WriteLine(c.Nome + " possui condições físicas");
			return true;
		}
	}

	//SubsistemaClasseB
	class ExamePsicotecnico
	{
		public bool possuiCondicoesMentais(Candidato c)
		{
			Console.WriteLine(c.Nome + " possui condições mentais");
			return true;
		}
	}

	//SubsistemaClasseC
	class ProvaPratica
	{
		public bool passou(Candidato c)
		{
			Console.WriteLine(c.Nome + " passou na prova prática");
			return true;
		}
	}

	//SubsistemaClasseD
	class PretensaoSalarial
	{
		public bool ehPossivel(Candidato c)
		{
			if(c.PretensaoSalarial > 7000 && c.PretensaoSalarial <
11000)
			{
				Console.WriteLine("Salário de " + c.Nome + " pode
ser pago");
				return true;
			}
			else
				return false;
		}
	}

	class Candidato
	{
		private string nome;
		private double pretensaoSalarial;
```

```
            public Candidato(string nome, double pretensaoSalarial)
            {
                  this.nome = nome;
                  this.pretensaoSalarial = pretensaoSalarial;
            }

            public string Nome
            {
                  get{ return nome;}
            }

            public double PretensaoSalarial
            {
                  get{ return pretensaoSalarial;}
            }
      }

      //Fachada
      class Contratacao
      {
            private ExameMedico exameMedico = new ExameMedico();
            private ExamePsicotecnico examePsicotecnico = new
ExamePsicotecnico();
            private ProvaPratica provaPratica = new ProvaPratica();
            private PretensaoSalarial pretensaoSalarial = new
PretensaoSalarial();

            //Verifica possibilidade de contratacao do candidato
            public bool seraContratado(Candidato c)
            {
                  Console.WriteLine("Verificando contratacao de " + c.Nome);

                  bool seraContratado = true;

                  if(!exameMedico.possuiCondicoesFisicas(c))
                        seraContratado = false;
                  if(!examePsicotecnico.possuiCondicoesMentais(c))
                        seraContratado = false;
                  if(!provaPratica.passou(c))
                        seraContratado = false;
                  if(!pretensaoSalarial.ehPossivel(c))
                        seraContratado = false;

                  return seraContratado;
            }
      }
}
```

3.2.5. Peso Pena (Flyweight)

Intenção

Usa compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina, de maneira eficiente. (Gamma, p.187)

Diagrama

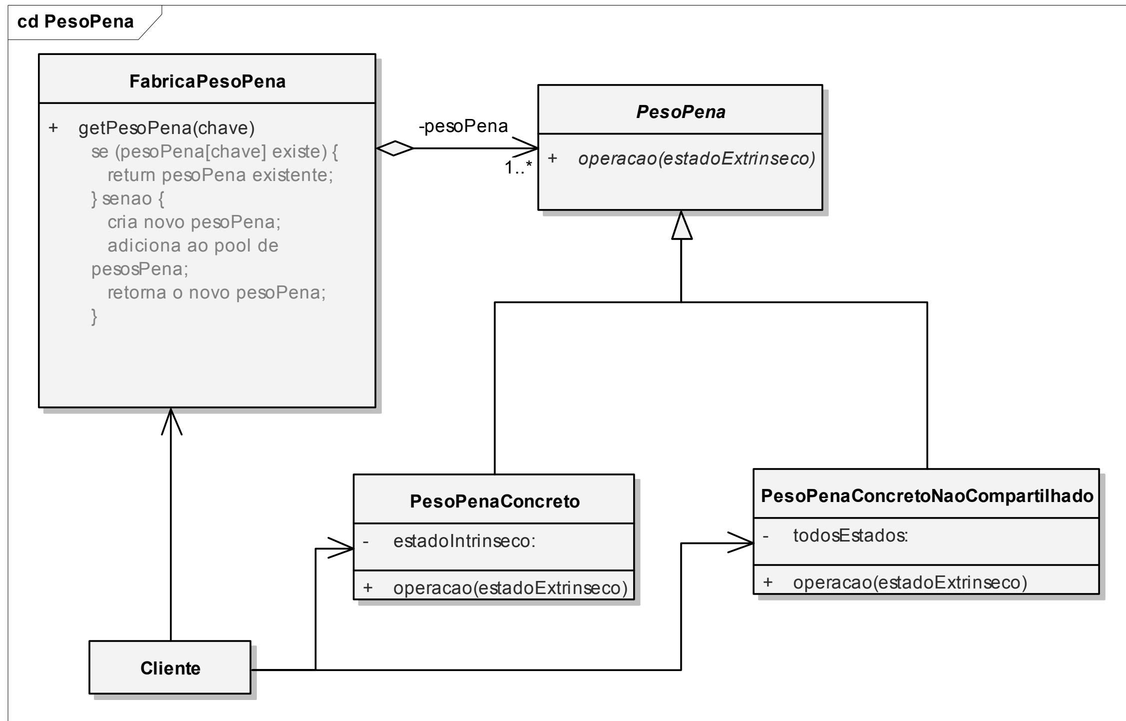

Figura 11 – Diagrama de Classes – Padrão Peso Pena

Explicações

O código abaixo demonstra o padrão Peso-Pena onde um número relativamente pequeno de objetos Caracter são compartilhados muitas vezes por um documento que possui potencialmente muitos caracteres.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Estrutural.PesoPena
{
	class Principal
	{
		static void Main()
		{
			// Constrói um documento com texto
			string documento = "AAZZBBZB";
			char[] caracteres = documento.ToCharArray();

			FabricaCaracter f = new FabricaCaracter();

			// estado extrínseco
			int tamanho = 10;

			// Para cada caracter, usa um objeto pesopena
			foreach (char c in caracteres)
			{
				tamanho++;
```

```
                        Caracter caracter = f.getCaracter(c);
                        caracter.Mostra(tamanho);
                }
        }
}

// "FabricaPesoPena"
class FabricaCaracter
{
        private Hashtable caracteres = new Hashtable();

        public Caracter getCaracter(char chave)
        {
                Caracter caracter = caracteres[chave] as Caracter;
                if (caracter == null)
                {
                        switch (chave)
                        {
                                case 'A': caracter = new CaracterA(); break;
                                case 'B': caracter = new CaracterB(); break;
                                        //...
                                case 'Z': caracter = new CaracterZ(); break;
                        }
                        caracteres.Add(chave, caracter);
                }
                return caracter;
        }
}

// "PesoPena"
abstract class Caracter
{
        protected char simbolo;
        protected int largura;
        protected int altura;
        protected int aclive;
        protected int declive;
        protected int tamanho;

        public abstract void Mostra(int tamanho);
}

// "PesoPenaConcreto"
class CaracterA : Caracter
{
        // Construtor
        public CaracterA()
        {
                this.simbolo = 'A';
                this.altura = 100;
                this.largura = 120;
                this.aclive = 70;
                this.declive = 0;
        }

        public override void Mostra(int tamanho)
        {
                this.tamanho = tamanho;
                Console.WriteLine(this.simbolo + " (tamanho " +
this.tamanho + ")");
        }
}

// "PesoPenaConcreto"
class CaracterB : Caracter
{
        // Construtor
        public CaracterB()
```

```
		{
			this.simbolo = 'B';
			this.altura = 100;
			this.largura = 140;
			this.aclive = 72;
			this.declive = 0;
		}

		public override void Mostra(int tamanho)
		{
			this.tamanho = tamanho;
			Console.WriteLine(this.simbolo + " (tamanho " +
this.tamanho + ")");
		}

	}

	// ... C, D, E, etc.

	// "PesoPenaConcreto"
	class CaracterZ : Caracter
	{
		// Construtor
		public CaracterZ()
		{
			this.simbolo = 'Z';
			this.altura = 100;
			this.largura = 100;
			this.aclive = 68;
			this.declive = 0;
		}

		public override void Mostra(int tamanho)
		{
			this.tamanho = tamanho;
			Console.WriteLine(this.simbolo + " (tamanho " +
this.tamanho + ")");
		}
	}
}
```

3.2.6. Ponte (Bridge)

Intenção

Segundo Gamma (p. 151) o padrão Ponte *separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.*

Diagrama

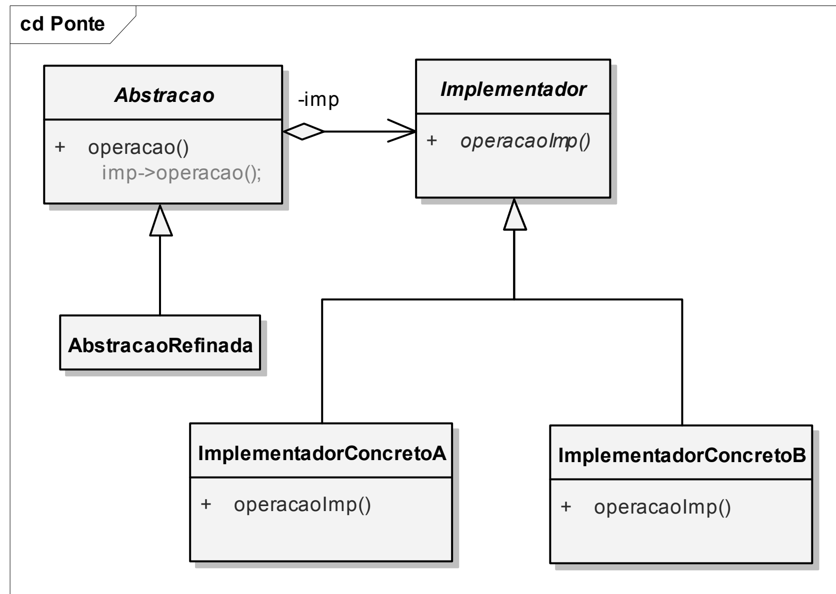

Figura 12 – Diagrama de Classes – Padrão Ponte

Explicações

O código abaixo demonstra o padrão Ponte onde uma abstração de Objeto de Negócios é desacoplado da implementação no Objeto de Dados. A implementação do Objeto de Dados pode evoluir dinamicamente sem modificar nenhum cliente.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Estrutural.Ponte
{
	class Principal
	{
		static void Main()
		{
			// Cria AbstracaoRefinada
			Selecao selecao = new Selecao("Brasil");

			// Seta ImplementadorConcreto
			selecao.Dados = new DadosSelecao();

			// Trabalha com a Ponte
			selecao.Mostra();
			selecao.Proximo();
			selecao.Mostra();
			selecao.Proximo();
			selecao.Mostra();
			selecao.Novo("Alex");

			selecao.MostraTodos();
		}
	}
```

```
// "Abstracao"
class BaseJogadores
{
	private ObjetoDados objetoDados;
	protected string grupo;

	public BaseJogadores(string grupo)
	{
		this.grupo = grupo;
	}

	// Propriedade
	public ObjetoDados Dados
	{
		set{ objetoDados = value; }
		get{ return objetoDados; }
	}

	public virtual void Proximo()
	{
		objetoDados.ProximoRegistro();
	}

	public virtual void Anterior()
	{
		objetoDados.RegistroAnterior();
	}

	public virtual void Novo(string nome)
	{
		objetoDados.NovoRegistro(nome);
	}

	public virtual void Excluir(string nome)
	{
		objetoDados.ExcluiRegistro(nome);
	}

	public virtual void Mostra()
	{
		objetoDados.MostraRegistro();
	}

	public virtual void MostraTodos()
	{
		Console.WriteLine("Grupo de Jogadores: " + grupo);
		objetoDados.MostraTodosRegistros();
	}
}

// "AbstracaoRefinada"
class Selecao : BaseJogadores
{
	// Construtor
	public Selecao(string grupo) : base(grupo)
	{
	}

	public override void MostraTodos()
	{
		// Adiciona linhas separadoras
		Console.WriteLine();
		Console.WriteLine ("------------------------");
		base.MostraTodos();
		Console.WriteLine ("------------------------");
	}
}
```

```
	// "Implementador"
	abstract class ObjetoDados
	{
		public abstract void ProximoRegistro();
		public abstract void RegistroAnterior();
		public abstract void NovoRegistro(string nome);
		public abstract void ExcluiRegistro(string nome);
		public abstract void MostraRegistro();
		public abstract void MostraTodosRegistros();
	}

	// "ImplementadorConcreto"
	class DadosSelecao : ObjetoDados
	{
		private ArrayList selecao = new ArrayList();
		private int atual = 0;

		public DadosSelecao()
		{
			// Carregados de um banco de Dados
			selecao.Add("Ronaldinho");
			selecao.Add("Ronaldo");
			selecao.Add("Robinho");
			selecao.Add("Cacá");
			selecao.Add("Adriano");
		}

		public override void ProximoRegistro()
		{
			if (atual <= selecao.Count - 1)
				atual++;
		}

		public override void RegistroAnterior()
		{
			if (atual > 0)
				atual--;
		}

		public override void NovoRegistro(string nome)
		{
			selecao.Add(nome);
		}

		public override void ExcluiRegistro(string nome)
		{
			selecao.Remove(nome);
		}

		public override void MostraRegistro()
		{
			Console.WriteLine(selecao[atual]);
		}

		public override void MostraTodosRegistros()
		{
			foreach (string nome in selecao)
				Console.WriteLine(" " + nome);
		}
	}
}
```

3.2.7. Procurador (Proxy)

Intenção

Gamma (p. 198) apresenta a definição do padrão Procurador como fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto, porém Braude (270) exemplifica melhor o propósito do padrão como evitar execução desnecessária de funcionalidade cara, de maneira transparente para os clientes.

Para tanto, Braude (p. 271) resume a implementação do padrão como *interpor uma classe que é acessada em vez daquela com funcionalidade cara*.

Diagrama

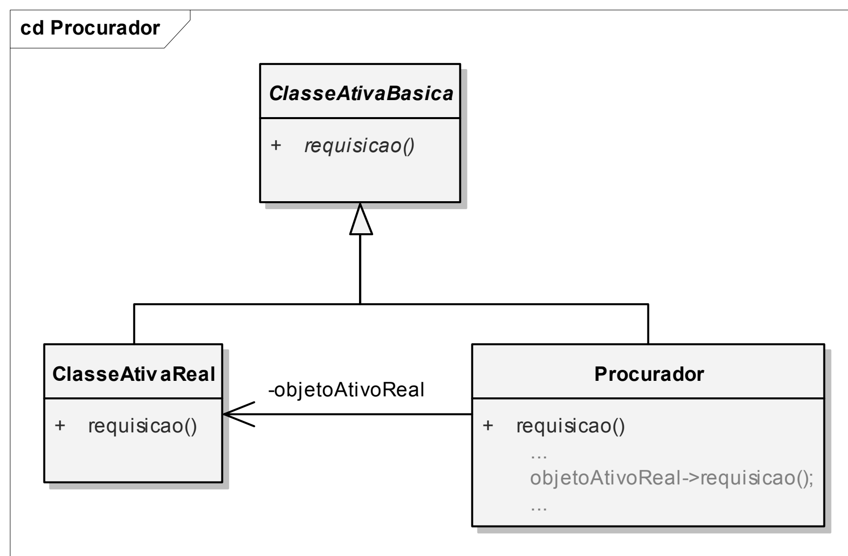

Figura 13 – Diagrama de Classes – Padrão Procurador

Explicações

O código abaixo demonstra o padrão Procurador sendo utilizado na representação de um objeto Matemático por um objeto ProcuradorMatemático.

Código

```
using System;

namespace TCC.MP.Estrutural.Procurador
{
	class Principal
	{
		static void Main()
		{
			// Cria o procuradorMatematico
			ProcuradorMatematico p = new ProcuradorMatematico();

			// Fas as Contas
```

```
                        Console.WriteLine("4 + 2 = " + p.Adicao(4, 2));
                        Console.WriteLine("4 - 2 = " + p.Subtracao(4, 2));
                        Console.WriteLine("4 * 2 = " + p.Multiplicacao(4, 2));
                        Console.WriteLine("4 / 2 = " + p.Divisao(4, 2));
                }
        }

        // "ClasseAtivaBasica, implementada na forma de interface"
        public interface IMatematica
        {
                double Adicao(double x, double y);
                double Subtracao(double x, double y);
                double Multiplicacao(double x, double y);
                double Divisao(double x, double y);
        }

        // "ClasseAtivaReal"
        class Matematica : IMatematica
        {
                public double Adicao(double x, double y){return x + y;}
                public double Subtracao(double x, double y){return x - y;}
                public double Multiplicacao(double x, double y){return x * y;}
                public double Divisao(double x, double y){return x / y;}
        }

        // "objeto Procurador"
        class ProcuradorMatematico : IMatematica
        {
                Matematica matematica;

                public ProcuradorMatematico()
                {
                        matematica = new Matematica();
                }

                public double Adicao(double x, double y)
                {
                        return matematica.Adicao(x,y);
                }
                public double Subtracao(double x, double y)
                {
                        return matematica.Subtracao(x,y);
                }
                public double Multiplicacao(double x, double y)
                {
                        return matematica.Multiplicacao(x,y);
                }
                public double Divisao(double x, double y)
                {
                        return matematica.Divisao(x,y);
                }
        }
}
```

3.3. Comportamentais

3.3.1. Cadeia de Responsabilidade (Chain of Responsibility)

Intenção

Para Gamma (p. 284) a intenção do padrão Cadeia de Responsabilidade é *evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores passando a solicitação ao longo da cadeia até que um objeto a trate.*

Para Braude (p. 355) o padrão permite que um conjunto de objetos atenda a uma solicitação, apresentando assim uma interface simples às aplicações-cliente.

Braude resume a implementação do projeto, *vinculando os objetos de uma cadeia via agregação, permitindo que cada um realize uma responsabilidade, passando a solicitação adiante.*

Diagrama

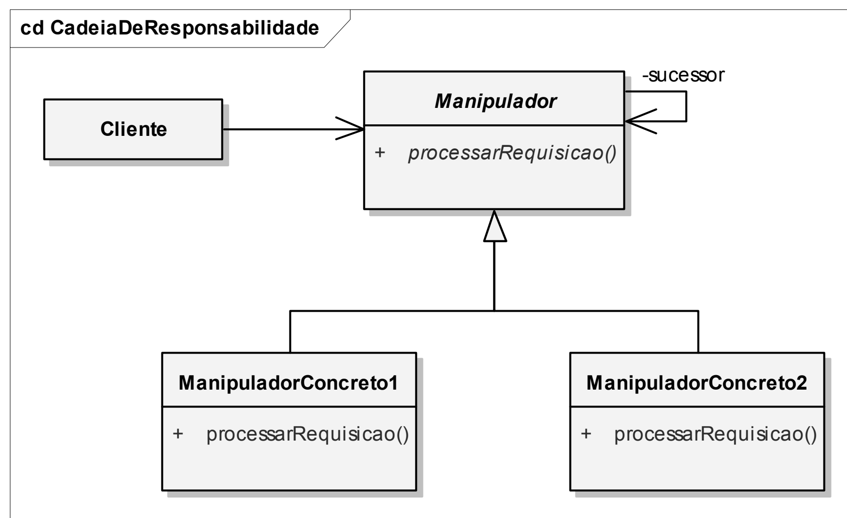

Figura 14 – Diagrama de Classes – Padrão Cadeia de Responsabilidade

Explicações

O código abaixo demonstra o padrão Cadeia de Responsabilidade sendo utilizado em um programa de solicitação de compra à diretoria de uma empresa.

Vários gerentes e executivos ligados entre si podem responder a uma solicitação ou repassar a mesma ao seu superior (sucessor) na cadeia. Cada membro da cadeia pode ter suas regras específicas de quais compras eles podem ou não aprovar.

Código

```
using System;

namespace TCC.MP.Comportamental.CadeiaDeResponsabilidade
{
	class Principal
	{
		static void Main()
		{
			// Configura a Cadeia de Responsabilidade
			Diretor Jose = new Diretor();
			VicePresidente Silva = new VicePresidente();
			Presidente Correa = new Presidente();
			Jose.setSucessor(Silva);
			Silva.setSucessor(Correa);

			// Gera e processa as solicitações de Compra
			Compra p = new Compra(2034, 350.00, "Suprimentos");
			Jose.processarSolicitacao(p);

			p = new Compra(2035, 32590.10, "Projeto X");
			Jose.processarSolicitacao(p);

			p = new Compra(2036, 122100.00, "Projeto Y");
			Jose.processarSolicitacao(p);
		}
	}

	// "Manipulador"
	abstract class Aprovador
	{
		protected Aprovador successor;

		public void setSucessor(Aprovador successor)
		{
			this.successor = successor;
		}

		public abstract void processarSolicitacao(Compra compra);
	}

	// "ManipuladorConcreto"
	class Diretor : Aprovador
	{
		public override void processarSolicitacao(Compra compra)
		{
			if (compra.Quantia < 10000.0)
			{
				Console.WriteLine("{0} aprovou solicitação # {1}",
this.GetType().Name, compra.Numero);
			}
			else if (successor != null)
			{
				successor.processarSolicitacao(compra);
			}
		}
	}
```

```
	// "ManipuladorConcreto"
	class VicePresidente : Aprovador
	{
		public override void processarSolicitacao(Compra compra)
		{
			if (compra.Quantia < 25000.0)
			{
				Console.WriteLine("{0} aprovou solicitação # {1}",
this.GetType().Name, compra.Numero);
			}
			else if (successor != null)
			{
				successor.processarSolicitacao(compra);
			}
		}
	}

	// "ManipuladorConcreto"
	class Presidente : Aprovador
	{
		public override void processarSolicitacao(Compra compra)
		{
			if (compra.Quantia < 100000.0)
			{
				Console.WriteLine("{0} aprovou solicitação # {1}",
this.GetType().Name, compra.Numero);
			}
			else
			{
				Console.WriteLine("Solicitação #{0} necessita de
uma reunião dos executivos!", compra.Numero);
			}
		}
	}

	// Detalhes da Solicitação
	class Compra
	{
		private int numero;
		private double quantia;
		private string finalidade;

		// Construtor
		public Compra(int numero, double quantia, string finalidade)
		{
			this.numero = numero;
			this.quantia = quantia;
			this.finalidade = finalidade;
		}

		// Propriedades
		public double Quantia
		{
			get{ return quantia; }
			set{ quantia = value; }
		}
		public string Finalidade
		{
			get{ return finalidade; }
			set{ finalidade = value; }
		}
		public int Numero
		{
			get{ return numero; }
			set{ numero = value; }
		}
	}
}
```

3.3.2. Comando (Command)

Intenção

Para Gamma (p. 222) a intenção do padrão Comando é *encapsular uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre(log) solicitações e suporte operações que podem ser desfeitas.*

Diagrama

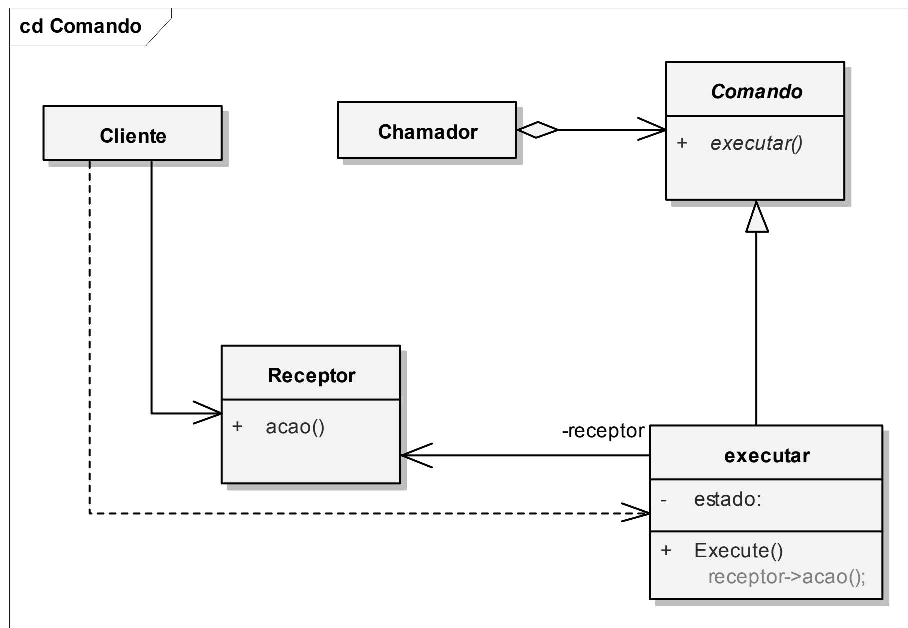

Figura 15 – Diagrama de Classes – Padrão Comando

Explicações

O código abaixo demonstra o padrão Comando sendo utilizando em um sistema de calculadora matemática simples onde um número ilimitado de funções desfazer e refazer podem ser executadas.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Comando
{
	class Principal
	{
		static void Main()
		{
```

```
                    // Cria o usuário e deixa ele computar
                    Usuario usuario = new Usuario();

                    usuario.computar('+', 100);
                    usuario.computar('-', 50);
                    usuario.computar('*', 10);
                    usuario.computar('/', 2);

                    // Desfaz 4 comandos
                    usuario.desfazer(4);

                    // Refaz 4 comandos
                    usuario.refazer(3);
             }
      }

      // "Comando"
      abstract class Comando
      {
             public abstract void executar();
             public abstract void desfazerExecucao();
      }

      // "ComandoConcreto"
      class ComandoCalculadora : Comando
      {
             char operador;
             int operando;
             Calculadora calculadora;

             // Construtor
             public ComandoCalculadora(Calculadora calculadora, char
operador, int operando)
             {
                    this.calculadora = calculadora;
                    this.operador = operador;
                    this.operando = operando;
             }

             public char Operador
             {
                    set{ operador = value; }
             }

             public int Operando
             {
                    set{ operando = value; }
             }

             public override void executar()
             {
                    calculadora.Operacao(operador, operando);
             }

             public override void desfazerExecucao()
             {
                    calculadora.Operacao(desfazer(operador), operando);
             }

             // Função de auxílio privada
             private char desfazer(char operador)
             {
                    char desfazer;
                    switch(operador)
                    {
                           case '+': desfazer = '-'; break;
                           case '-': desfazer = '+'; break;
                           case '*': desfazer = '/'; break;
```

```
                        case '/': desfazer = '*'; break;
                        default : desfazer = ' '; break;
                }
                return desfazer;
        }
}

// "Receptor"
class Calculadora
{
        private int atual = 0;

        public void Operacao(char operador, int operando)
        {
                switch(operador)
                {
                        case '+': atual += operando; break;
                        case '-': atual -= operando; break;
                        case '*': atual *= operando; break;
                        case '/': atual /= operando; break;
                }
                Console.WriteLine("Valor atual = {0,3} (executado {1}
{2})",
                        atual, operador, operando);
        }
}

// "Chamador"
class Usuario
{
        // Inicializadores
        private Calculadora calculadora = new Calculadora();
        private ArrayList comandos = new ArrayList();

        private int atual = 0;

        public void refazer(int niveis)
        {
                Console.WriteLine("\n---- refazer {0} niveis ", niveis);
                // Realiza operacoes refazer
                for (int i = 0; i < niveis; i++)
                {
                        if (atual < comandos.Count - 1)
                        {
                                Comando comando = comandos[atual++] as
Comando;
                                comando.executar();
                        }
                }
        }

        public void desfazer(int niveis)
        {
                Console.WriteLine("\n---- desfazer {0} niveis ", niveis);
                // Realiza operacoes desfazer
                for (int i = 0; i < niveis; i++)
                {
                        if (atual > 0)
                        {
                                Comando comando = comandos[--atual] as
Comando;
                                comando.desfazerExecucao();
                        }
                }
        }

        public void computar(char operador, int operando)
        {
```

```
                    // Cria a operacao do comando e executa-a
                    Comando comando = new ComandoCalculadora(calculadora,
operador, operando);
                    comando.executar();

                    // Adiciona comando à lista desfazer
                    comandos.Add(comando);
                    atual++;
              }
       }
}
```

3.3.3. Estado (State)

Intenção

Para Gamma (p. 284) o padrão Estado *permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.*

Braude (p. 346) define o propósito do padrão como *fazer com que um objeto comporte-se de uma forma determinada por seu estado.*

Diagrama

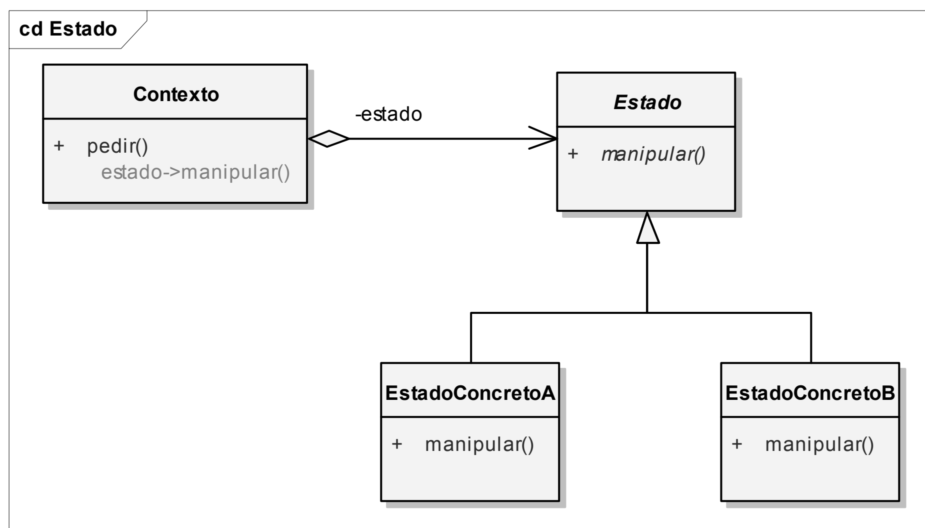

Figura 16 – Diagrama de Classes – Padrão Estado

Explicações

O código abaixo demonstra a utilização do padrão Estado que permite que um Interruptor tenha comportamentos diferentes dependendo do seu estado atual. A diferença nos comportamentos é delegada aos objetos chamados Ligado e Desligado.

Por exemplo, o interruptor deve ter comportamentos diferentes quando chega um pedido para ligá-lo, quando seu estado já estiver ligado e quando estiver desligado.

Código

```
using System;

namespace TCC.MP.Comportamental.Estado
{
	class Principal
	{
		static void Main()
		{
			Interruptor interruptor = new Interruptor();

			//Liga e desliga o interruptor
			interruptor.ligar();
			interruptor.desligar();
			interruptor.desligar();
			interruptor.ligar();
			interruptor.ligar();
			interruptor.desligar();
		}
	}

	// "Estado"
	abstract class Estado
	{
		protected Interruptor interruptor;
		protected bool aceso;

		// Propriedades
		public Interruptor Interruptor
		{
			get{ return interruptor; }
			set{ interruptor = value; }
		}

		public bool Aceso
		{
			get{ return aceso; }
			set{ aceso = value; }
		}

		public abstract void ligar();
		public abstract void desligar();
	}

	// "EstadoConcreto"

	// Estado para Interruptor Ligado
	class Ligado : Estado
	{
		// Sobrecarga de Construtores
		public Ligado(Estado estado) : this(estado.Interruptor)
		{
		}

		public Ligado(Interruptor interruptor)
		{
			aceso = true;
			this.interruptor = interruptor;
		}

		public override void ligar()
```

```
        {
                aceso = true;
                ChecaModificacaoEstado();
        }

        public override void desligar()
        {
                aceso = false;
                ChecaModificacaoEstado();
        }

        private void ChecaModificacaoEstado()
        {
                if (aceso == false)
                {
                        interruptor.Estado = new Desligado(this);
                        Console.WriteLine("Desligando Interruptor");
                }
                else
                        Console.WriteLine("Interruptor já ligado!");
        }
}

// "EstadoConcreto"

// Estado para Interruptor Desligado
class Desligado : Estado
{
        // Sobrecarga de Construtores
        public Desligado(Estado estado) : this(estado.Interruptor)
        {
        }

        public Desligado(Interruptor interruptor)
        {
                aceso = false;
                this.interruptor = interruptor;
        }

        public override void ligar()
        {
                aceso = true;
                ChecaModificacaoEstado();
        }

        public override void desligar()
        {
                aceso = false;
                ChecaModificacaoEstado();
        }

        private void ChecaModificacaoEstado()
        {
                if (aceso == true)
                {
                        interruptor.Estado = new Ligado(interruptor);
                        Console.WriteLine("Ligando Interruptor");
                }
                else
                        Console.WriteLine("Interruptor já desligado!");
        }
}


// "Contexto"
class Interruptor
{
        private Estado estado;
```

```
            // Construtor
            public Interruptor()
            {
                  estado = new Desligado(this);
            }

            // Propriedades
            public bool Ligado
            {
                  get{ return estado.Aceso; }
            }

            public Estado Estado
            {
                  get{ return estado; }
                  set{ estado = value; }
            }

            public void ligar()
            {
                  estado.ligar();
            }

            public void desligar()
            {
                  estado.desligar();
            }
      }
}
```

3.3.4. Estratégia (Strategy)

Intenção

Para Gamma (p. 292) o padrão Estratégia *define uma família de algoritmos, encapsula cada um deles e faze-os intercambiáveis. O padrão permite que o algoritmo varie independentemente dos clientes que o utilizam.*

Diagrama

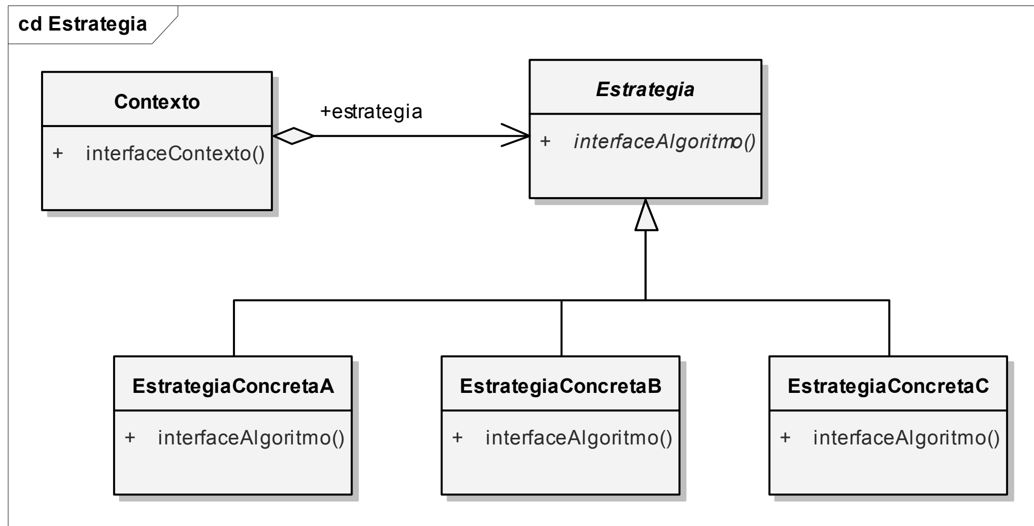

Figura 17 – Diagrama de Classes – Padrão Estratégia

Explicações

O código abaixo demonstra o padrão Estratégia que encapsula algoritmos de ordenamento na forma de objetos de ordenação. Isso permite ao cliente dinamicamente trocar o método de ordenamento entre Quicksort, Shellsort e Mergesort.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Estrategia
{
	class Principal
	{
		static void Main()
		{
			// Cria o contenxo seguindo os ordenamentos
			ListaOrdenada registrosEstudante = new ListaOrdenada();

			registrosEstudante.adiciona("João");
			registrosEstudante.adiciona("José");
			registrosEstudante.adiciona("Maria");
			registrosEstudante.adiciona("Pedro");
			registrosEstudante.adiciona("Ana");

			registrosEstudante.setEstrategiaOrdenamento(new
QuickSort());
			registrosEstudante.ordena();

			registrosEstudante.setEstrategiaOrdenamento(new
ShellSort());
			// nao sera visto diferenca pois ShellSort nao foi
implementado
			registrosEstudante.ordena();

			registrosEstudante.setEstrategiaOrdenamento(new
```

```
MergeSort());
                    // nao sera visto diferenca pois MergeSort nao foi
implementado
                    registrosEstudante.ordena();
              }
       }

       // "Estrategia"
       abstract class EstrategiaOrdenamento
       {
              public abstract void ordena(ArrayList lista);
       }

       // "EstrategiaConcreta"
       class QuickSort : EstrategiaOrdenamento
       {
              public override void ordena(ArrayList lista)
              {
                    lista.Sort(); // Padrão é Quicksort
                    Console.WriteLine("Lista - QuickSort ");
              }
       }

       // "EstrategiaConcreta"
       class ShellSort : EstrategiaOrdenamento
       {
              public override void ordena(ArrayList lista)
              {
                    //lista.ShellSort(); nao-implementado
                    Console.WriteLine("Lista - ShellSort ");
              }
       }

       // "EstrategiaConcreta"
       class MergeSort : EstrategiaOrdenamento
       {
              public override void ordena(ArrayList lista)
              {
                    //lista.MergeSort(); nao-implementado
                    Console.WriteLine("Lista - MergeSort ");
              }
       }

       // "Contexto"
       class ListaOrdenada
       {
              private ArrayList lista = new ArrayList();
              private EstrategiaOrdenamento estrategiaOrdenamento;

              public void setEstrategiaOrdenamento(EstrategiaOrdenamento
estrategiaOrdenamento)
              {
                    this.estrategiaOrdenamento = estrategiaOrdenamento;
              }

              public void adiciona(string nome)
              {
                    lista.Add(nome);
              }

              public void ordena()
              {
                    estrategiaOrdenamento.ordena(lista);

                    // Mostra resultados
                    foreach (string nome in lista)
                    {
                          Console.WriteLine(" " + nome);
```

```
                    }
                    Console.WriteLine();
                }
            }
}
```

3.3.5. Gabarito (Template)

Intenção

Para Gamma (p. 301) o padrão Gabarito *define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O padrão Gabarito permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.*

Braude (p. 366) define o propósito do padrão como *permitir variantes em tempo de execução de um algoritmo.*

Diagrama

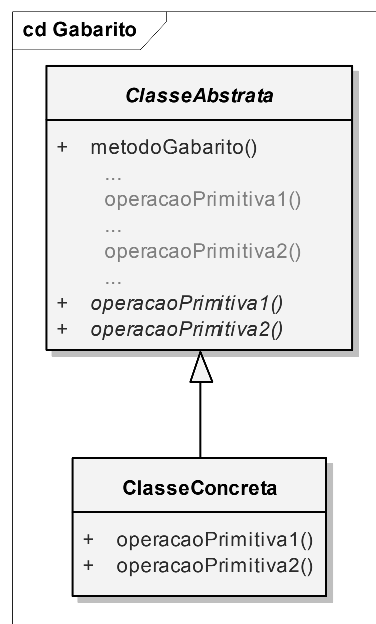

Figura 18 – Diagrama de Classes – Padrão Gabarito

Explicações

O código abaixo demonstra o padrão Gabarito sendo utilizado no método executa(), o qual provê o esqueleto de uma seqüência de chamadas a métodos. Os métodos conecta() e disconecta() são comuns e reaproveitados nos Objetos de Acesso a Dados herdeiros da classe abstrata. Porém os métodos seleciona() e processa() são específicos de cada Objeto de Acesso

a Dados, portanto são métodos abstratos da classe abstrata, e devem ser reescritos nas classes herdeiras.

Código[*] [†]

```
using System;
using System.Data;
using System.Data.OleDb;

namespace TCC.MP.Comportamental.Gabarito
{
	class Principal
	{
		static void Main()
		{
			DataAccessObject dao;

			dao = new Clientes();
			dao.executa();

			dao = new Produtos();
			dao.executa();
		}
	}

	// "ClasseAbstrata"
	abstract class DataAccessObject
	{
		protected string stringConexao;

		protected DataSet dataSet;

		public virtual void conecta()
		{
			// Tenha certeza que o caminho do banco está correto
			stringConexao = "provider=Microsoft.JET.OLEDB.4.0; data
source=c:\\cadastros.mdb";
		}

		public abstract void seleciona();
		public abstract void processa();

		public virtual void disconecta()
		{
			stringConexao = "";
		}

		// O "Método Gabarito"
		public void executa()
		{
			conecta();
			seleciona();
			processa();
			disconecta();
```

[*] Para o exemplo funcionar, é necessário que exista um banco de dados access com nome cadastros.mdb salvo no diretório c:/. Este banco de dados deve contar pelos menos uma tabela cliente com coluna nome e uma tabela produtos com coluna descricao.

[†] O nome DataAccessObject não foi traduzido para o português, por se tratar de um padrão de projeto não abordado por Gamma, porém utilizado em outros catálogos de padrões como os padrões para Java (J2EE), para maiores informações:
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

```
		}
	}

	// "ClasseConcreta"
	class Clientes : DataAccessObject
	{
		public override void seleciona()
		{
			string sql = "SELECT nome FROM cliente";
			OleDbDataAdapter dataAdapter = new OleDbDataAdapter(sql,
stringConexao);

			dataSet = new DataSet();
			dataAdapter.Fill(dataSet, "cliente");
		}

		public override void processa()
		{
			Console.WriteLine("Clientes ---- ");

			DataTable dataTable = dataSet.Tables["cliente"];
			foreach (DataRow row in dataTable.Rows)
			{
				Console.WriteLine(row["nome"]);
			}
			Console.WriteLine();
		}
	}

	class Produtos : DataAccessObject
	{
		public override void seleciona()
		{
			string sql = "SELECT descricao FROM produto";
			OleDbDataAdapter dataAdapter = new OleDbDataAdapter(sql,
stringConexao);

			dataSet = new DataSet();
			dataAdapter.Fill(dataSet, "produto");
		}

		public override void processa()
		{
			Console.WriteLine("Produtos ---- ");
			DataTable dataTable = dataSet.Tables["produto"];
			foreach (DataRow row in dataTable.Rows)
			{
				Console.WriteLine(row["descricao"]);
			}
			Console.WriteLine();
		}
	}
}
```

3.3.6. Interpretador (Interpreter)

Intenção

Gamma (p. 231) define a seguinte intenção para o padrão Interpretador:

"Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem".

Braude (p. 322) define mais claramente o propósito do padrão como "interpretar expressões escritas em uma gramática formal".

Diagrama

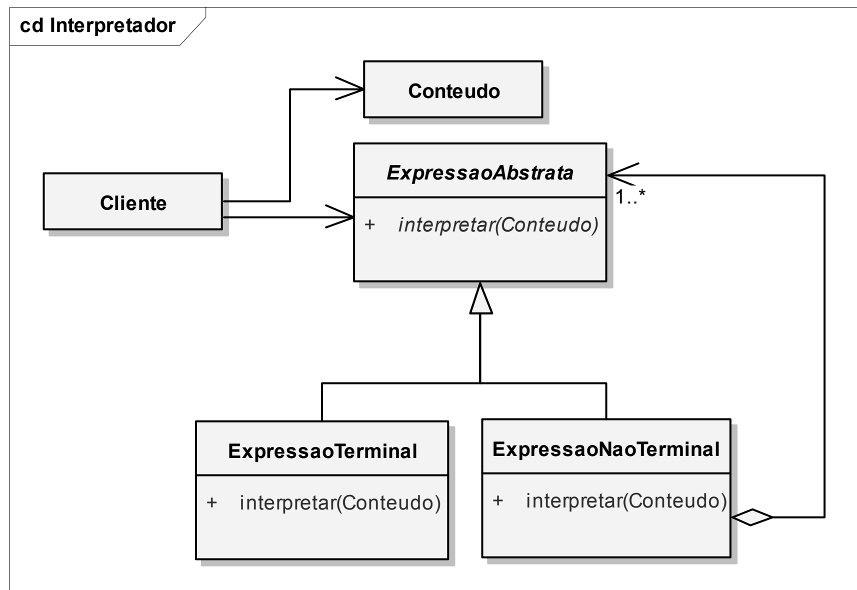

Figura 19 – Diagrama de Classes – Padrão Interpretador

Explicações

O código abaixo demonstra o padrão Interpretador, o qual usa uma gramática definida (Numerais Romanos) que provê um interpretador que processa um conteúdo passado como parâmetro (Número Decimal).

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Interpretador
{
	class Principal
	{
```

```
		static void Main()
		{
			string romano = "MCMXXVIII";
			Conteudo conteudo = new Conteudo(romano);

			// Constrói a árvore de análise gramatical
			ArrayList arvore = new ArrayList();
			arvore.Add(new ExpressaoMilhares());
			arvore.Add(new ExpressaoCentenas());
			arvore.Add(new ExpressaoDezenas());
			arvore.Add(new ExpressaoUnidades());

			// interpretar
			foreach (Expressao exp in arvore)
			{
				exp.interpretar(conteudo);
			}

			Console.WriteLine("{0} = {1}", romano, conteudo.Saida);
		}
	}

	// "Conteudo"
	class Conteudo
	{
		private string entrada;
		private int saida;

		// Construtor
		public Conteudo(string entrada)
		{
			this.entrada = entrada;
		}

		// Propriedades
		public string Entrada
		{
			get{ return entrada; }
			set{ entrada = value; }
		}

		public int Saida
		{
			get{ return saida; }
			set{ saida = value; }
		}
	}

	// "ExpressaoAbstrata"
	abstract class Expressao
	{
		public void interpretar(Conteudo conteudo)
		{
			if (conteudo.Entrada.Length == 0)
				return;

			if (conteudo.Entrada.StartsWith(Nove()))
			{
				conteudo.Saida += (9 * Multiplicador());
				conteudo.Entrada = conteudo.Entrada.Substring(2);
			}
			else if (conteudo.Entrada.StartsWith(Quatro()))
			{
				conteudo.Saida += (4 * Multiplicador());
				conteudo.Entrada = conteudo.Entrada.Substring(2);
			}
			else if (conteudo.Entrada.StartsWith(Cinco()))
			{
```

```
                        conteudo.Saida += (5 * Multiplicador());
                        conteudo.Entrada = conteudo.Entrada.Substring(1);
                  }

                  while (conteudo.Entrada.StartsWith(Um()))
                  {
                        conteudo.Saida += (1 * Multiplicador());
                        conteudo.Entrada = conteudo.Entrada.Substring(1);
                  }
            }

            public abstract string Um();
            public abstract string Quatro();
            public abstract string Cinco();
            public abstract string Nove();
            public abstract int Multiplicador();
      }

      // Milhares checa pelo Numeral Romano M
      // "ExpressaoTerminal"
      class ExpressaoMilhares : Expressao
      {
            public override string Um() { return "M"; }
            public override string Quatro(){ return " "; }
            public override string Cinco(){ return " "; }
            public override string Nove(){ return " "; }
            public override int Multiplicador() { return 1000; }
      }

      // Centenas checa pelos Numerais Romanos C, CD, D e CM
      // "ExpressaoTerminal"
      class ExpressaoCentenas : Expressao
      {
            public override string Um() { return "C"; }
            public override string Quatro(){ return "CD"; }
            public override string Cinco(){ return "D"; }
            public override string Nove(){ return "CM"; }
            public override int Multiplicador() { return 100; }
      }

      // Dezenas checa pelos Numerais Romanos X, XL, L e XC
      // "ExpressaoTerminal"
      class ExpressaoDezenas : Expressao
      {
            public override string Um() { return "X"; }
            public override string Quatro(){ return "XL"; }
            public override string Cinco(){ return "L"; }
            public override string Nove(){ return "XC"; }
            public override int Multiplicador() { return 10; }
      }

      // Unidades checa por I, II, III, IV, V, VI, VI, VII, VIII, IX
      // "ExpressaoTerminal"
      class ExpressaoUnidades : Expressao
      {
            public override string Um() { return "I"; }
            public override string Quatro(){ return "IV"; }
            public override string Cinco(){ return "V"; }
            public override string Nove(){ return "IX"; }
            public override int Multiplicador() { return 1; }
      }
}
```

3.3.7. Iterador (Iterator)

Intenção

Gamma (p. 244): Fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

Diagrama

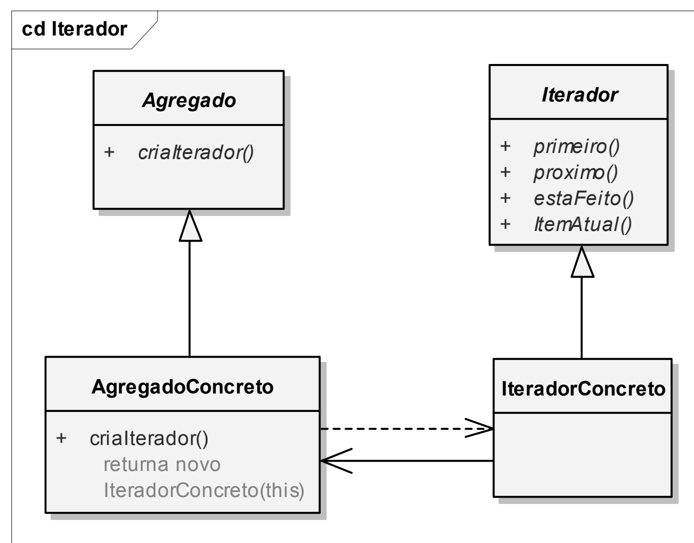

Figura 20 – Diagrama de Classes – Padrão Iterador

Explicações

O código abaixo demonstra o padrão Iterador que é usado para iterar sobre uma coleção de itens e pular um número específico de itens a cada iteração

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Iterador
{
	class Principal
	{
		static void Main()
		{
			// Constrói a coleção
			Colecao colecao = new Colecao();
			colecao[0] = new Item("Item 0");
			colecao[1] = new Item("Item 1");
			colecao[2] = new Item("Item 2");
			colecao[3] = new Item("Item 3");
			colecao[4] = new Item("Item 4");
			colecao[5] = new Item("Item 5");
			colecao[6] = new Item("Item 6");
			colecao[7] = new Item("Item 7");
			colecao[8] = new Item("Item 8");
```

```
                // Cria Iterador
                Iterador iterador = new Iterador(colecao);

                // Pula todos os outros itens
                iterador.Passo = 2;

                Console.WriteLine("Iterando sobre colecao:");

                for(Item item = iterador.primeiro(); !iterador.EstaFeito;
item = iterador.proximo())
                {
                        Console.WriteLine(item.Nome);
                }
        }
    }

    class Item
    {
        string nome;

        // Construtor
        public Item(string nome)
        {
                this.nome = nome;
        }

        // Propriedade
        public string Nome
        {
                get{ return nome; }
        }
    }

    // "Agregado, implementado como interface"
    interface IColecaoAbstrata
    {
        Iterador criaIterador();
    }

    // "AgregadoConcreto"
    class Colecao : IColecaoAbstrata
    {
        private ArrayList itens = new ArrayList();

        public Iterador criaIterador()
        {
                return new Iterador(this);
        }

        // Propriedade
        public int Contador
        {
                get{ return itens.Count; }
        }

        // Indexador
        public object this[int indice]
        {
                get{ return itens[indice]; }
                set{ itens.Add(value); }
        }
    }

    // "Iterador, implementado como interface"
    interface IIteradorAbstrato
    {
        Item primeiro();
```

```
            Item proximo();
            bool EstaFeito{ get; }
            Item ItemAtual{ get; }
    }

    // "IteradorConcreto"
    class Iterador : IIteradorAbstrato
    {
            private Colecao colecao;
            private int atual = 0;
            private int passo = 1;

            // Construtor
            public Iterador(Colecao colecao)
            {
                    this.colecao = colecao;
            }

            public Item primeiro()
            {
                    atual = 0;
                    return colecao[atual] as Item;
            }

            public Item proximo()
            {
                    atual += passo;
                    if (!EstaFeito)
                            return colecao[atual] as Item;
                    else
                            return null;
            }

            // Propriedades
            public int Passo
            {
                    get{ return passo; }
                    set{ passo = value; }
            }

            public Item ItemAtual
            {
                    get
                    {
                            return colecao[atual] as Item;
                    }
            }

            public bool EstaFeito
            {
                    get
                    {
                            return atual >= colecao.Contador ? true : false;
                    }
            }
    }
}
```

3.3.8. Mediador (Mediator)

Intenção

Gamma (p. 257) resume a intenção do Padrão Mediador como:

Define um objeto que encapsula como um conjunto de objetos interage. O Mediador promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.

Braude (p. 342) define o propósito do padrão simplificadamente como *evitar referências entre objetos dependentes*.

Diagrama

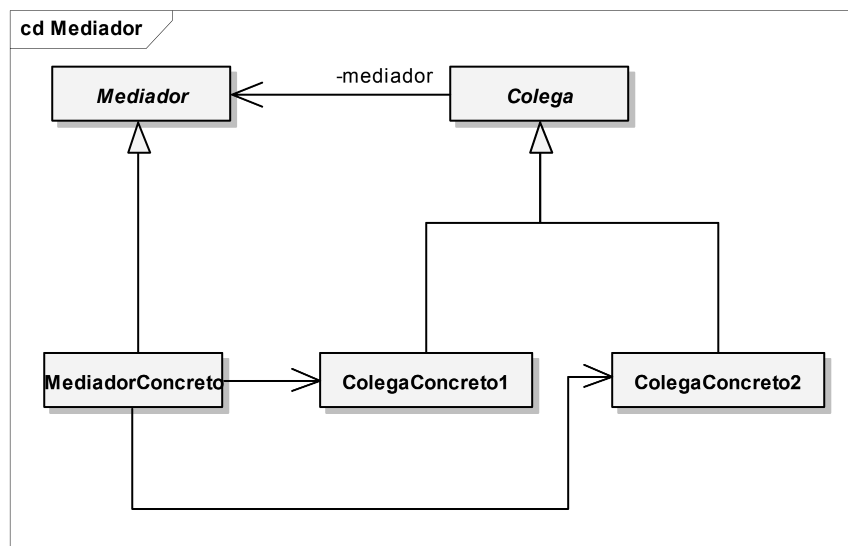

Figura 21 – Diagrama de Classes – Padrão Mediador

Explicações

O código abaixo demonstra o padrão Mediador facilitando a comunicação via acoplamento fraco entre diferentes Participantes registrados em uma SalaDeBatePapo. A SalaDeBatePapo é o ponto central por onde passam todas as comunicações. No exemplo abaixo, apenas comunicações um-a-um são implementadas na SalaDeBatePapo, porém, pode ser implementadas comunicações tipo broadcast (um-a-muitos).

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Mediador
{
	class Principal
	{
		static void Main()
		{
			// Cria sala01
			SalaDeBatePapo sala01 = new SalaDeBatePapo();

			// Cria e registra os participantes na sala
			Participante Jorge = new Homem("Jorge");
			Participante Paulo = new Homem("Paulo");
			Participante Tiago = new Homem("Tiago");
			Participante Marcos = new Homem("Marcos");
			Participante Ana = new Mulher("Ana");

			sala01.registra(Jorge);
			sala01.registra(Paulo);
			sala01.registra(Tiago);
			sala01.registra(Marcos);
			sala01.registra(Ana);

			// Participantes Batendo Papo
			Ana.envia ("Marcos", "Olá Marcos!");
			Paulo.envia ("Tiago", "E aí? Quais são as novidades?");
			Tiago.envia ("Jorge", "Seu time é muito ruim!");
			Paulo.envia ("Marcos", "Essa sala só tem uma mulher!!!");
			Marcos.envia ("Ana", "Vamos sair hoje?");
		}
	}

	// "Mediador"
	abstract class SalaDeBatePapoAbstrata
	{
		public abstract void registra(Participante participante);
		public abstract void envia(string de, string para, string
mensagem);
	}

	// "MediadorConcreto"
	class SalaDeBatePapo : SalaDeBatePapoAbstrata
	{
		private Hashtable participantes = new Hashtable();

		public override void registra(Participante participante)
		{
			if (participantes[participante.Nome] == null)
				participantes[participante.Nome] = participante;

			participante.SalaDeBatePapo = this;
		}

		public override void envia(string de, string para, string
mensagem)
		{
			Participante pto = (Participante)participantes[para];
			if (pto != null)
				pto.recebe(de, mensagem);
		}
	}

	// "ColegaAbstrato"
```

```
	class Participante
	{
		private SalaDeBatePapo sala01;
		private string nome;

		// Construtor
		public Participante(string nome)
		{
			this.nome = nome;
		}

		// Propriedades
		public string Nome
		{
			get{ return nome; }
		}

		public SalaDeBatePapo SalaDeBatePapo
		{
			set{ sala01 = value; }
			get{ return sala01; }
		}

		// Métodos
		public void envia(string para, string mensagem)
		{
			sala01.envia(nome, para, mensagem);
		}

		public virtual void recebe(string de, string mensagem)
		{
			Console.WriteLine("{0} para {1}: '{2}'", de, Nome,
mensagem);
		}
	}

	//"ColegaConcreto1"
	class Homem : Participante
	{
		// Construtor
		public Homem(string nome) : base(nome)
		{
		}

		public override void recebe(string de, string mensagem)
		{
			Console.Write("Para um Homem: ");
			base.recebe(de, mensagem);
		}
	}

	//"ColegaConcreto2"
	class Mulher : Participante
	{
		// Construtor
		public Mulher(string nome) : base(nome)
		{
		}

		public override void recebe(string de, string mensagem)
		{
			Console.Write("Para uma Mulher: ");
			base.recebe(de, mensagem);
		}
	}
}
```

3.3.9. Observador (Observer)

Intenção

Para Gamma (p. 274) o padrão Observador *define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.*

Braude (p. 346) define o propósito do padrão simplificadamente como *providenciar que um conjunto de objetos seja afetado por um único objeto*.

Diagrama

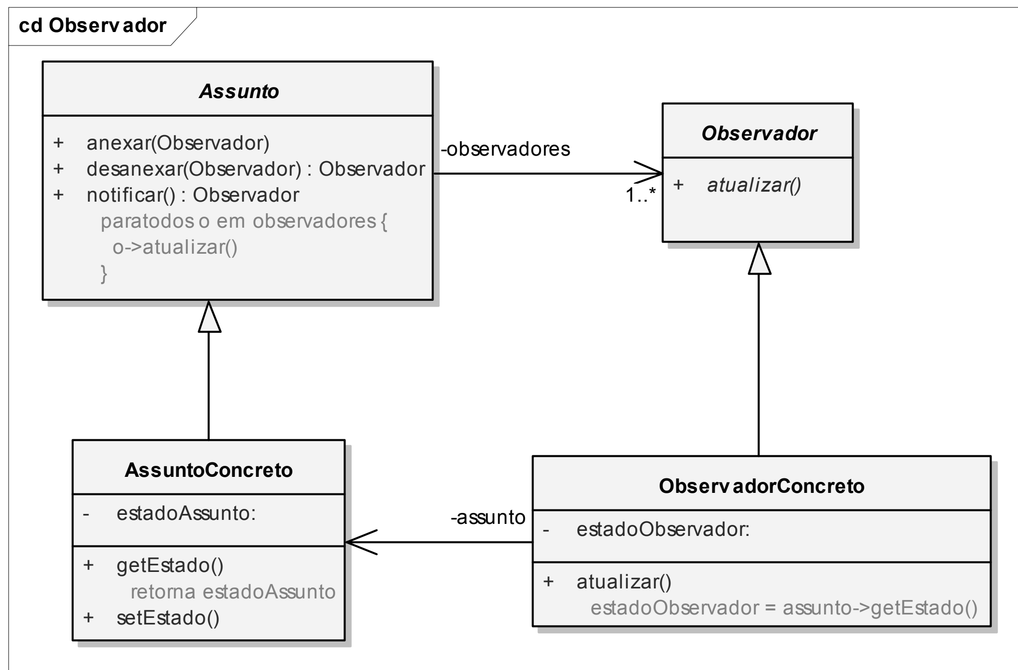

Figura 22 – Diagrama de Classes – Padrão Observador

Explicações

O código abaixo demonstra a utilização do padrão Observador onde um PostoDeGasolina registrado (anexado) a uma Distribuidora recebe uma notificação sempre que a Distribuidora altera o preço do litro da gasolina.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Observador
```

```
{
	class Principal
	{
		static void Main()
		{
			// Cria os postos de gasolina
			PostoDeGasolina p1 = new PostoDeGasolina("Posto do
Papai");
			PostoDeGasolina p2 = new PostoDeGasolina("Posto da
Mamãe");

			// Cria a distribuidora e anexa os postos a ela
			Distribuidora petrobras = new Distribuidora("Petrobrás",
2.15);
			petrobras.anexar(p1);
			petrobras.anexar(p2);

			// Muda o preco da distribuidora que notifica os postos
			petrobras.PrecoLitroGasolina = 2.18;
			petrobras.PrecoLitroGasolina = 2.22;
			petrobras.PrecoLitroGasolina = 2.25;
			petrobras.PrecoLitroGasolina = 2.28;
		}
	}

	// "Assunto"
	abstract class DistribuidoraAbstrata
	{
		protected string nome;
		protected double preco;
		private ArrayList postos = new ArrayList();

		// Construtor
		public DistribuidoraAbstrata(string nome, double preco)
		{
			this.nome = nome;
			this.preco = preco;
		}

		public void anexar(PostoDeGasolina posto)
		{
			postos.Add(posto);
		}

		public void desanexar(PostoDeGasolina posto)
		{
			postos.Remove(posto);
		}

		public void notificar()
		{
			foreach (PostoDeGasolina posto in postos)
			{
				posto.atualizar(this);
			}
			Console.WriteLine("");
		}

		// Propriedades
		public double PrecoLitroGasolina
		{
			get{ return preco; }
			set
			{
				preco = value;
				notificar();
			}
		}
```

```
            public string Nome
            {
                  get{ return nome; }
                  set{ nome = value; }
            }
      }

      // "AssuntoConcreto"
      class Distribuidora : DistribuidoraAbstrata
      {
            // Construtor
            public Distribuidora(string nome, double preco) : base(nome,
preco)
            {
            }
      }

      // "Observador"
      interface IPosto
      {
            void atualizar(DistribuidoraAbstrata distribuidora);
      }

      // "ObservadorConcreto"
      class PostoDeGasolina : IPosto
      {
            private string nome;
            private Distribuidora distribuidora;

            // Construtor
            public PostoDeGasolina(string nome)
            {
                  this.nome = nome;
            }

            public void atualizar(DistribuidoraAbstrata distribuidora)
            {
                  Console.WriteLine("{0} notificado pela {1} a mudar o preco
para {2:C}", nome, distribuidora.Nome, distribuidora.PrecoLitroGasolina);
            }

            // Propriedade
            public Distribuidora Distribuidora
            {
                  get{ return distribuidora; }
                  set{ distribuidora = value; }
            }
      }
}
```

3.3.10. Recordação (Memento)

Intenção

Para Gamma (p. 266) a intenção do padrão Recordação é *capturar e externalizar um estado interno de um objeto, sem violar o encapsulamento, de modo que o mesmo possa posteriormente ser restaurado para este estado.*

Diagrama

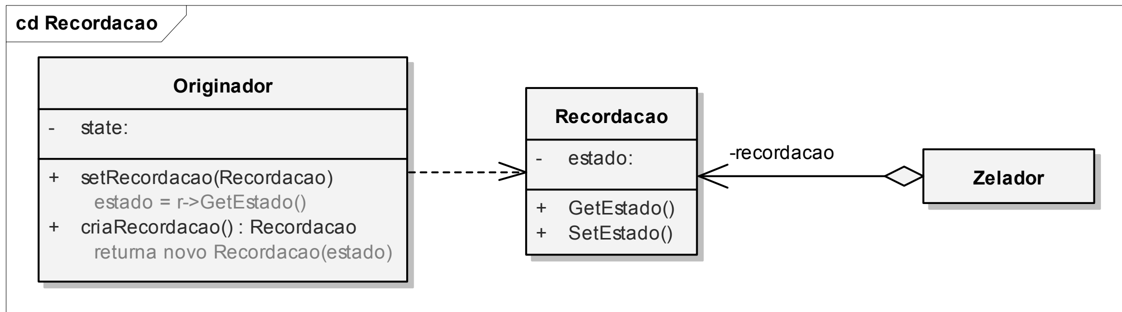

Figura 23 – Diagrama de Classes – Padrão Recordação

Explicações

O código abaixo demonstra o padrão Recordação onde o que temporariamente salva e depois restaura o estado interno de um Prospecto de Venda.

Código

```
using System;

namespace TCC.MP.Comportamental.Recordacao
{
	class Principal
	{
		static void Main()
		{
			ProspectoVendas s = new ProspectoVendas();
			s.Nome = "Marcos Paulo";
			s.Telefone = "(031) 1234-5678";
			s.Orcamento = 25000.0;

			// Salva estado interno
			MemoriaProspecto m = new MemoriaProspecto();
			m.Recordacao = s.SalvaRecordacao();

			// Continua modificando originador
			s.Nome = "Marques Corrêa";
			s.Telefone = "(031) 8765-4321";
			s.Orcamento = 100000.0;

			// Restaura estado salvo
			s.RestauraRecordacao(m.Recordacao);
		}
	}

	// "Originador"
	class ProspectoVendas
	{
		private string nome;
		private string telefone;
		private double orcamento;

		// Propriedades
		public string Nome
		{
			get{ return nome; }
			set
			{
```

```
                        nome = value;
                        Console.WriteLine("Nome: " + nome);
                }
        }

        public string Telefone
        {
                get{ return telefone; }
                set
                {
                        telefone = value;
                        Console.WriteLine("Telefone: " + telefone);
                }
        }

        public double Orcamento
        {
                get{ return orcamento; }
                set
                {
                        orcamento = value;
                        Console.WriteLine("Orcamento: " + orcamento);
                }
        }

        public Recordacao SalvaRecordacao()
        {
                Console.WriteLine("\nSalvando Estado --\n");
                return new Recordacao(nome, telefone, orcamento);
        }

        public void RestauraRecordacao(Recordacao recordacao)
        {
                Console.WriteLine("\nRestaurando Estado --\n");
                this.Nome = recordacao.Nome;
                this.Telefone = recordacao.Telefone;
                this.Orcamento = recordacao.Orcamento;
        }
    }

    // "Recordacao"

    class Recordacao
    {
        private string nome;
        private string telefone;
        private double orcamento;

        // Construtor
        public Recordacao(string nome, string telefone, double
orcamento)
        {
                this.nome = nome;
                this.telefone = telefone;
                this.orcamento = orcamento;
        }

        // Propriedades
        public string Nome
        {
                get{ return nome; }
                set{ nome = value; }
        }

        public string Telefone
        {
                get{ return telefone; }
                set{ telefone = value; }
```

```
            }

            public double Orcamento
            {
                  get{ return orcamento; }
                  set{ orcamento = value; }
            }
      }

      // "Zelador"
      class MemoriaProspecto
      {
            private Recordacao recordacao;

            // Propriedade
            public Recordacao Recordacao
            {
                  set{ recordacao = value; }
                  get{ return recordacao; }
            }
      }
}
```

3.3.11. Visitante (Visitor)

Intenção

Para Gamma (p. 305) o padrão Visitante *representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O Visitante permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.*

Diagrama

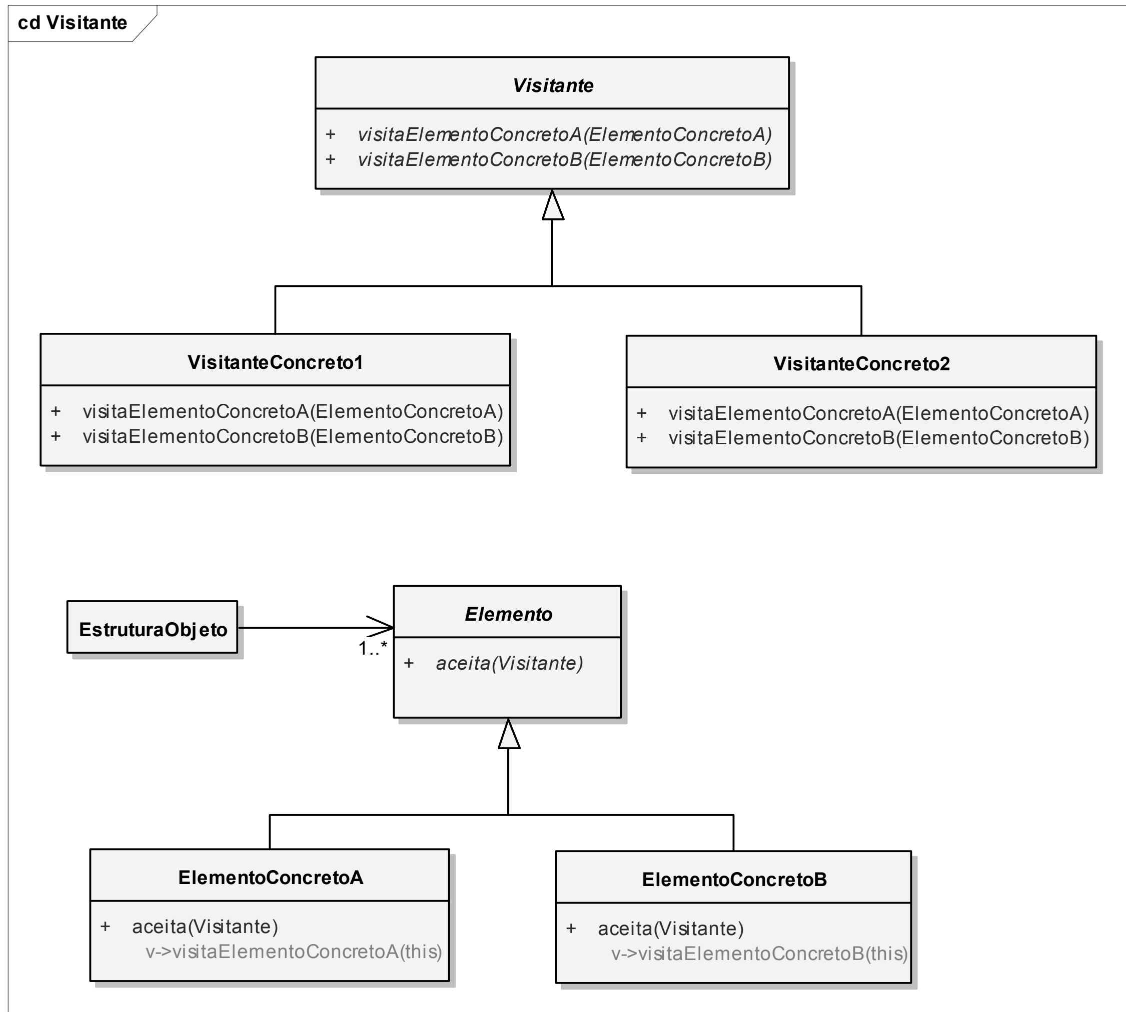

Figura 24 – Diagrama de Classes – Padrão Visitante

Explicações

O código abaixo demonstra o padrão Visitante onde dois objetos (SalarioVisitante e FeriasVisitante) “visitam” a lista de Empregados e executam as mesmas operações em cada empregado. Os dois objetos visitantes definem diferentes operações – um ajusta os dias de férias e o outro o salário.

Código

```
using System;
using System.Collections;

namespace TCC.MP.Comportamental.Visitante
{
	class MainApp
	{
		static void Main()
		{
```

```
                    // Configura coleção de empregados
                    Empregados e = new Empregados();
                    e.anexa(new Secretario());
                    e.anexa(new Diretor());
                    e.anexa(new Presidente());

                    // Empregados sao 'visitados'
                    e.aceita(new SalarioVisitante());
                    e.aceita(new FeriasVisitante());
             }
       }

       // "Visitante"
       interface IVisitante
       {
             void visitar(Elemento elemento);
       }

       // "VisitanteConcreto1"
       class SalarioVisitante : IVisitante
       {
             public void visitar(Elemento elemento)
             {
                    Empregado empregado = elemento as Empregado;

                    // Dá aumento de 10%
                    empregado.Salario *= 1.10;
                    Console.WriteLine("Novo salário do {0} {1}: {2:C}",
empregado.GetType().Name, empregado.Nome, empregado.Salario);
             }
       }

       // "VisitanteConcreto2"
       class FeriasVisitante : IVisitante
       {
             public void visitar(Elemento elemento)
             {
                    Empregado empregado = elemento as Empregado;

                    // Dá 3 dias Extras de Férias
                    empregado.DiasFerias += 3;
                    Console.WriteLine("Novos dias de férias do {0} {1}: {2}",
empregado.GetType().Name, empregado.Nome, empregado.DiasFerias);
             }
       }

       class Secretario : Empregado
       {
             // Construtor
             public Secretario() : base("João", 25000.0, 14)
             {
             }
       }

       class Diretor : Empregado
       {
             // Construtor
             public Diretor() : base("José", 35000.0, 16)
             {
             }
       }

       class Presidente : Empregado
       {
             // Construtor
             public Presidente() : base("Pedro", 45000.0, 21)
             {
             }
```

```
	}

	// "Elemento"
	abstract class Elemento
	{
		public abstract void aceita(IVisitante visitante);
	}

	// "ElementoConcreto"
	class Empregado : Elemento
	{
		string nome;
		double salario;
		int diasFerias;

		// Construtor
		public Empregado(string nome, double salario, int diasFerias)
		{
			this.nome = nome;
			this.salario = salario;
			this.diasFerias = diasFerias;
		}

		// Propriedades
		public string Nome
		{
			get{ return nome; }
			set{ nome = value; }
		}

		public double Salario
		{
			get{ return salario; }
			set{ salario = value; }
		}

		public int DiasFerias
		{
			get{ return diasFerias; }
			set{ diasFerias = value; }
		}

		public override void aceita(IVisitante visitante)
		{
			visitante.visitar(this);
		}
	}

	// "EstruturaObjeto"
	class Empregados
	{
		private ArrayList empregados = new ArrayList();

		public void anexa(Empregado empregado)
		{
			empregados.Add(empregado);
		}

		public void desanexa(Empregado empregado)
		{
			empregados.Remove(empregado);
		}

		public void aceita(IVisitante visitante)
		{
			foreach (Empregado e in empregados)
			{
				e.aceita(visitante);
```

```
                        }
                        Console.WriteLine();
                }
        }
}
```

3.4. Freqüência de Uso

O gráfico abaixo, baseado em dados do site Data & Object Factory, define a freqüência de uso de cada padrão de projeto.

O site citado acima define a freqüência em uma escala de 1 (baixo) a 5 (alto).

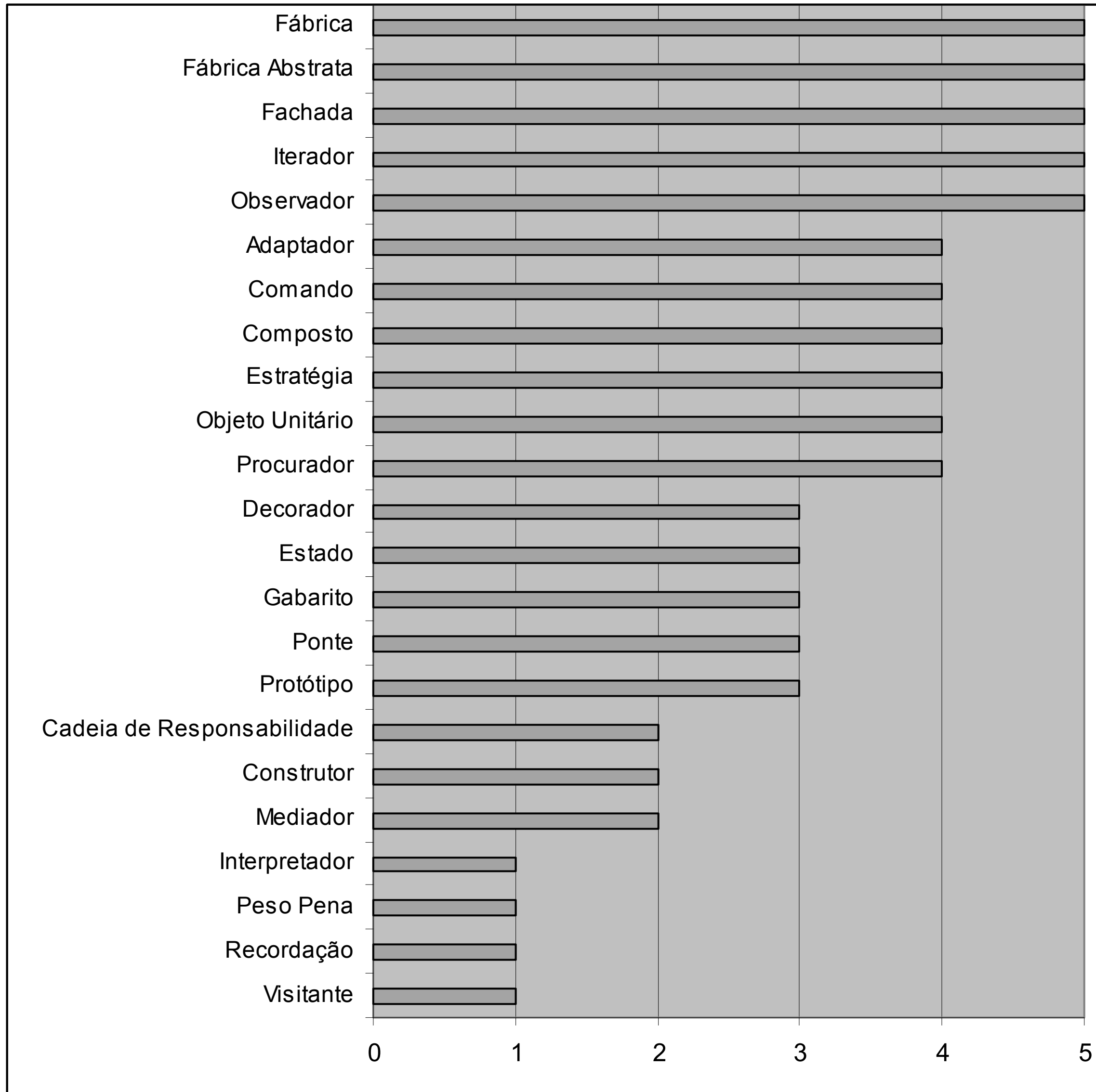

Figura 25 - Gráfico de Frequência de Uso dos Padrões de Projeto

3.5. Relacionamento entre os Padrões

A imagem abaixo, apresentada por Gamma et al. (2000, p. 27) apresenta os relacionamentos existentes entre os padrões de projeto.

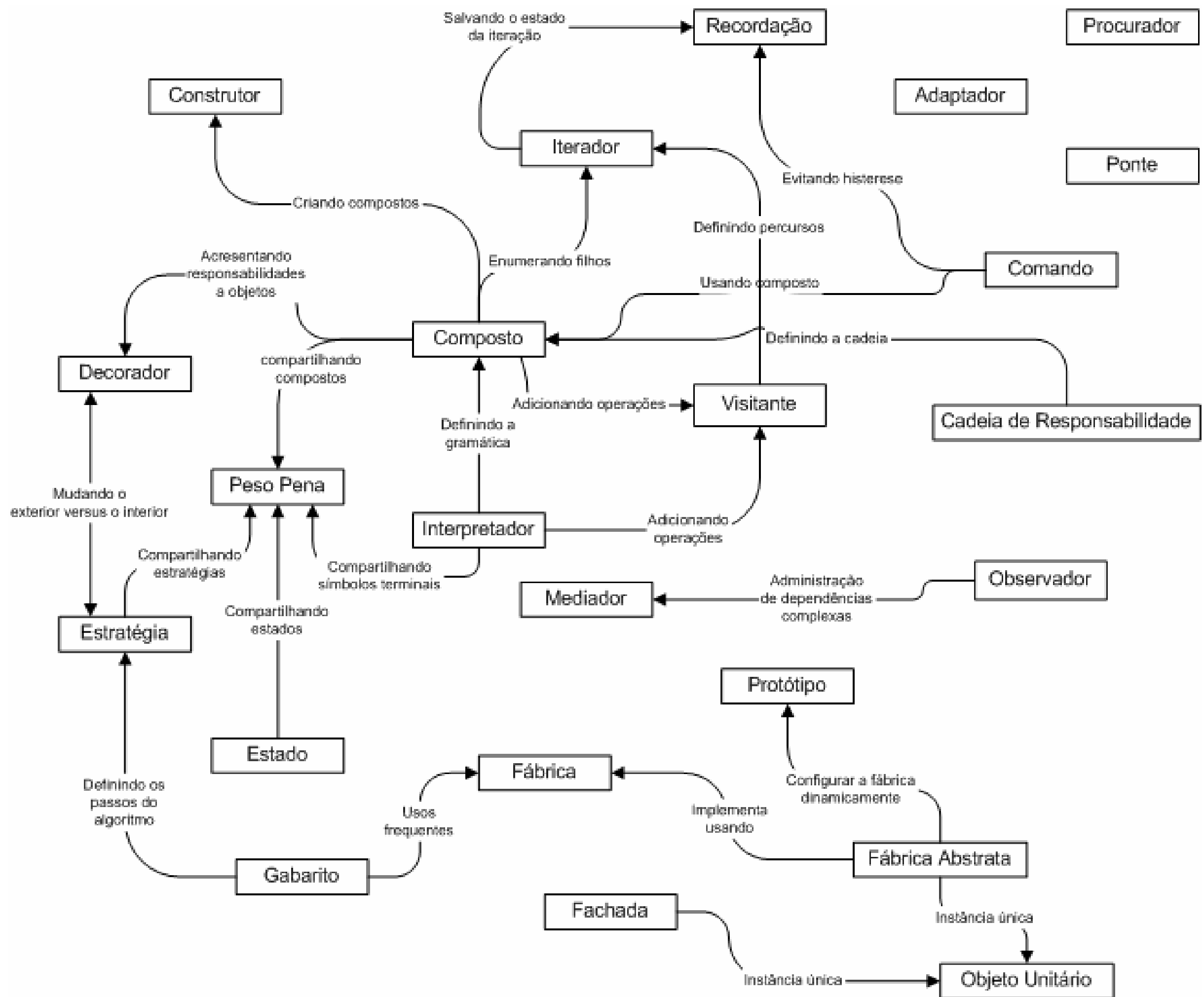

Figura 26 - Relacionamento entre Padrões de Projeto

4. CONCLUSÃO

Constatei ao longo do trabalho que os padrões de projeto apresentados pela GoF são extremamente úteis em um ambiente empresarial com vários projetistas, analistas e programadores utilizando cada um uma linguagem ou plataforma diferente, pois os padrões são genéricos e podem ser aplicados em áreas diversas, apenas com pequenas mudanças de sintaxe na forma como serão passados do modelo para a realidade necessária.

Os padrões além de servirem como um manual ou catálogo de boas práticas a serem seguidos pelos envolvidos no projeto de software da empresa, servem como um vocabulário comum entre os mesmos, como por exemplo “Vamos usar uma *Cadeia de Responsabilidade* aqui”, ou ainda “Nessa classe deveria ser utilizado um *Objeto Unitário*”.

Desde que se especifique quais os padrões de projeto estão ou estarão sendo utilizados em um projeto de software, eles ajudarão no entendimento de todos os envolvidos. Mesmo que o sistema esteja pronto, rodando e funcionando, os padrões podem ser utilizados como alvo de uma Refatoração[*] de código.

Em suma, os padrões de projeto, como o próprio nome diz, ajudarão um projetista, seja ele iniciante ou experiente, a projetar seus códigos mais padronizados, conseguindo um melhor reaproveitamento / reutilização do mesmo, tornando-se assim um melhor programador.

Foi constatado ainda que a linguagem C#.NET, com sua sintaxe simples, robusta, e completamente orientada a objetos auxilia na construção de um código simples e de fácil entendimento e aprendizado, tornando menos complexa a tarefa de tirar dos diagramas o projeto do software orientado a objetos e traduzí-los em uma linguagem de alto nível.

---

[*] Refatoração (do inglês *Refactoring*) é o processo de modificar um sistema de *software* para melhorar a estrutura interna do código sem alterar seu comportamento externo.

5. REFERÊNCIAS BIBLIOGRÁFICAS

AHMED, Mesbah, et al. **ASP.NET Guia do Desenvolvedor Web**. Rio de Janeiro: Alta Books.

BARBOSA, Antonio Carlos. **Programando em C# com .Net Framework**. São Paulo: Érica, 2002.

BRAUDE, Eric. **Projeto de Software: da programação à arquitetura: uma abordagem baseada em Java** – Porto Alegre: Bookman, 2005.

DEBONI, José Eduardo Zindel. **Modelagem orientada a objetos com a UML**. São Paulo: Futura, 2003.

GAMMA, Erich, et al. **Padrões de Projeto – Soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2000.

GUEDES, Gilleanes T. A. **UML – Uma Abordagem Prática**. São Paulo: Novatec, 2004.

LARMAN, Craig. **Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao Processo Unificado**. 2ª Ed. Porto Alegre: Bookman, 2004.

TURTSCHI, Adrian, et al. **C#.NET Guia do Desenvolvedor**. 2ª ED. Rio de Janeiro: Alta Books.

Sites

Object Management Group, 2001. **OMG Unified Modeling Language Specification**. Disponível em: <http://www.omg.org> Acesso: 26 nov. 2005

Data & Object Factory. Disponível em: <http://www.dofactory.com> Acesso: 27 nov. 2005