PADRÕES DE PROJETO – CONCEITOS E APLICAÇÕES

ANDRADE, Leandro Prado de; SOUZA, Avelino Megda Martins de; CAVALCANTE, José Elton Figueiredo; PRIMO, Lucas Leandro(1); REIS, José Cláudio de Sousa(2).
(1) Acadêmicos do oitavo período do Curso de Ciência da Computação, UNIFENAS, Alfenas.
(2) Professor do Curso de Ciência da Computação , UNIFENAS, Alfenas.

RESUMO:

O padrão de projeto descreve uma solução para determinado problema que pode ser aplicada inúmeras vezes. Porém a utilização desse padrão nunca será da mesma forma, deve ser adaptado para o problema em questão. Este trabalho tem como objetivo demonstrar a aplicação dos padrões de projeto no desenvolvimento de um software e obter componentes reutilizáveis com a utilização dos padrões. Foi utilizado para o desenvolvimento do trabalho o ambiente de desenvolvimento Visual Studio, linguagem de programação C#, banco de dados MySQL, conceitos de orientação a objetos e arquitetura em camadas. O trabalho foi feito desenvolvendo-se um software inicial sem a aplicação dos padrões. Posteriormente foi aplicada uma refatoração no código do software inicial aplicando-se os padrões de projeto. As funcionalidades dos padrões aplicados bem como a codificação necessária para utilização dos padrões são demonstradas neste trabalho. Conclui-se que com a aplicação dos padrões é possível obter componentes reutilizáveis. O trabalho permitiu a demonstração da aplicação dos padrões de projeto.

PALAVRAS CHAVE: padrões de projeto, aplicação, componentes reutilizáveis

ABSTRACT:

The design pattern describes a solution to a problem which can be applied numerous times. However, the use of this pattern will never be the same, it must be adapted to the problem at hand. This work aims to demonstrate the application of design patterns in the development of software and get reusable components with the use of standards. For the development of this work, the development environment Visual Studio, C # programming language, MySQL database, object oriented concepts and layered architecture were used. The work was done by developing an initial software without the application of standards. Later the initial software code was refactored by applying design patterns. The features as well as the standards applied to coding standards required for use are shown in this study. It is concluded that with the implementation of the standards it is possible to obtain reusable components. The study allowed the demonstration of the application of design patterns.

KEYWORDS: design patterns, implementation, reusable components

1. INTRODUÇÃO

1.1 Caracterização do problema em estudo

O padrão de projeto descreve uma solução para determinado ambiente que pode ser aplicada inúmeras vezes. Porém a utilização desse padrão nunca será da mesma forma.

Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil e também para que seja possível tornar o projeto reutilizável.

O padrão permite ao projetista fazer uso de uma solução pronta no desenvolvimento do sistema, tendo este que apenas adaptar a solução para o sistema em questão.

1.2 Objetivo

O objetivo do presente trabalho é demonstrar a aplicação dos padrões de projeto no desenvolvimento de um software e obter componentes reutilizáveis com a utilização dos padrões.

1.3 Hipóteses

Espera-se que com o uso dos padrões o código fique melhor estruturado e apresente componentes reutilizáveis

2. REFERENCIAL TEÓRICO

2.1 Padrões de projeto

Segundo Alexander (1979), um padrão pode ser caracterizado como uma regra de três partes que expressa uma relação entre um contexto, um problema e uma solução. Para projeto de software o contexto permite a compreensão do ambiente em que o problema reside

e qual solução poderia ser apropriada nesse ambiente. Um conjunto de requisitos influencia como o problema pode ser interpretado e como a solução pode ser efetivamente aplicada.

Coplien, em 2005, caracteriza um padrão de projeto eficaz como:

- Ele soluciona um problema;
- É um conceito comprovado;
- Uma solução não é óbvia;
- Descreve uma relação;
- Possui um componente humano significativo.

Os padrões de projeto podem ser divididos em padrões de criação, padrões estruturais e padrões comportamentais.

2.1.1 Padrões de criação

Segundo Gamma et. al. (1995) os padrões de criação abstraem o processo de instanciação. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto um padrão de criação de objeto delegará a instanciação para outro objeto.

2.1.2 Padrões estruturais

Gamma et. al. em 1995, afirma que os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classe utilizam a herança para compor interfaces ou implementações.

2.1.3 Padrões comportamentais

Os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidades entre objetos. Eles descrevem a comunicação entre objetos e classes e caracterizam fluxos de controle difíceis de seguir em tempo de execução. Também afastam o foco do fluxo de controle para permitir a concentração somente na maneira como os objetos são interconectados. (Gamma et.al., 1995).

3. MATERIAL E MÉTODOS

O trabalho foi desenvolvido em quatro etapas. Na primeira etapa foi realizado um levantamento bibliográfico para obter informações sobre como era realizado o desenvolvimento de software antes da utilização dos processos de desenvolvimento, conhecer os processos de desenvolvimento, e procurar na literatura conteúdo disponível referente aos padrões de projeto.

Na segunda etapa, foi realizada a seleção da tecnologia a ser utilizada. Foram utilizados para o desenvolvimento do software o ambiente de programação Microsoft Visual Studio 2012, a linguagem de programação C#, programação orientada a objetos e arquitetura em camadas.

Na terceira etapa, o software foi desenvolvido. Foi desenvolvida uma primeira versão do software utilizando boas práticas e programação e orientação a objetos, porém não serão utilizados os padrões de projeto. Posteriormente, uma segunda versão do software será desenvolvida, aplicando-se nesta versão os padrões de projeto.

Na quarta e última etapa, foram realizados os testes e comparativos no software. Após o sistema ser devidamente testado foram realizadas as comparações entre a primeira e a segunda versão do sistema. Foi verificado o quanto o código ficou mais legível, o quanto o código foi simplificado, se a segunda versão do software apresenta componentes reutilizáveis e o ganho na facilidade e eficiência na manutenção do sistema.

4. DESENVOLVIMENTO

4.1 Sistema Netmovies

O sistema Netmovies oferece os serviços de cadastro e vendas de filmes online, procurando oferecer com qualidade e facilidade o acesso aos filmes.

Desenvolveram-se duas versões do sistema Netmovies – uma sem a aplicação dos padrões de projeto e uma com a aplicação dos padrões de projeto.

Após o término da primeira versão do sistema, realizou-se um processo de refatoração com a finalidade de aplicar os três tipos de padrões de projeto: criacionais, estruturais e comportamentais.

4.2 Aplicação dos padrões criacionais

Os padrões de criacionais estão relacionados à criação de objetos.

4.2.1 Padrão Prototype

O padrão Prototype especifica os tipos de objeto a serem criados usando um protótipo de instância o qual permite que novos objetos sejam criados por meio da cópia deste protótipo.

Este padrão é usado quando um sistema tiver de ser independente de como os seus produtos são criados, compostos e representados.

Na primeira versão do projeto, o objeto tipodto era criado somente uma vez, não possibilitando a reutilização dos dados e a clonagem do objeto. Pode ser visualizada no código abaixo a instanciação dos objetos: tipodto e tipobll. O objeto tipobll contém os métodos com as operações sobre o banco de dados. O

objeto tipodto é responsável por armazenar os dados referentes a um tipo de filme. Observa-se também a atribuição de valores aos atributos do objeto tipodto e a operação de inserção no banco de dados passado como parâmetro o objeto tipodto.

```
            tipoBLL tipobll = new tipoBLL();
             tipoDTO tipodto = new tipoDTO();

             tipodto.NOME_TIPO = txtNometipo.Text;
             tipodto.DURACAO_TIPO = 
Convert.ToInt32(txtDuracao.Text);
             tipodto.VALOR_TIPO = 
Convert.ToDecimal(txtValor.Text);
             tipodto.DATA_CAD_TIPO = DateTime.Now;
             tipobll.INSERE(tipodto);
```

Para implementação do padrão Prototype criou-se a classe abstrata Prototype que serve de modelo para a classe Tipo, classe esta que possibilita a clonagem de objetos.

Pode-se observar no código abaixo, a classe Prototype com seus atributos, propriedades e o método abstrato Clone cujo tipo é o mesmo da classe. Nota-se também que os valores dos atributos são passados no método construtor.

```
public abstract class Prototype{
        int id_tipo, duracao_tipo;
        decimal valor_tipo;
        DateTime data_cad_tipo;
        string nome_tipo;
        public int ID_TIPO{
            get { return id_tipo; }
            set { id_tipo = value; }
        }
        public int DURACAO_TIPO{
            get { return duracao_tipo; }
            set { duracao_tipo = value; }
        }
        public decimal VALOR_TIPO{
            get { return valor_tipo; }
            set { valor_tipo = value; }
        }
        public DateTime DATA_CAD_TIPO{
            get { return data_cad_tipo; }
```

```
            set { data_cad_tipo = value; }
        }
        public string NOME_TIPO{
            get { return nome_tipo; }
            set { nome_tipo = value; }
        }
        public abstract Prototype Clone();
        public Prototype(int id, string nome, DateTime data,int
duracao,decimal valor) {
            ID_TIPO = id;
            NOME_TIPO = nome;
            DATA_CAD_TIPO = data;
            VALOR_TIPO = valor;
            DURACAO_TIPO = duracao;
        }}
```

Finalmente, criou-se a classe Tipo que implementa a classe abstrata Prototype e possibilita a clonagem dos objetos. Segue o código da classe Tipo:

```
public class Tipo : Prototype{
        public Tipo(int id, string nome, DateTime data, int
duracao, decimal valor) : base(id,nome,data,duracao,valor) {}
        public override Prototype Clone(){
            return (Prototype)this.MemberwiseClone();
        } }
```

Nota-se que os valores dos atributos da classe Tipo são passados no seu método construtor. O método Clone da classe Tipo tem como retorno um objeto da classe a ser clonada o qual é obtido por meio do método MemberwiseClone. O método MemberwiseClone é nativo na linguagem C#. Este método cria um novo objeto com os mesmos valores definidos na classe que realiza a chamada a este método.

Segue abaixo o código da utilização do padrão Prototype. Criou-se um objeto do tipo Tipo com o nome de tipodto, onde foram inseridos os valores no momento de sua instanciação. Feito isso, criou-se um novo objeto também do tipo Tipo com o nome de clone. Porém, este novo objeto é um clone do objeto tipodto e é obtido com a chamada ao método Clone.

```
            Tipo tipodto = new Tipo(id,txtNomecategoria.Text,
DateTime.Now, Convert.ToInt32(txtDuracao.Text),
Convert.ToDecimal(txtValor.Text));
```

```
Tipo clone = (Tipo)tipodto.Clone();
```

Na FIG. 1 está ilustrado o diagrama das classes utilizadas para implementação do padrão Protoype. Pode-se observar: a classe abstrata Prototype com seus atributos, propriedades e métodos; e a classe Tipo com os seus respectivos métodos.

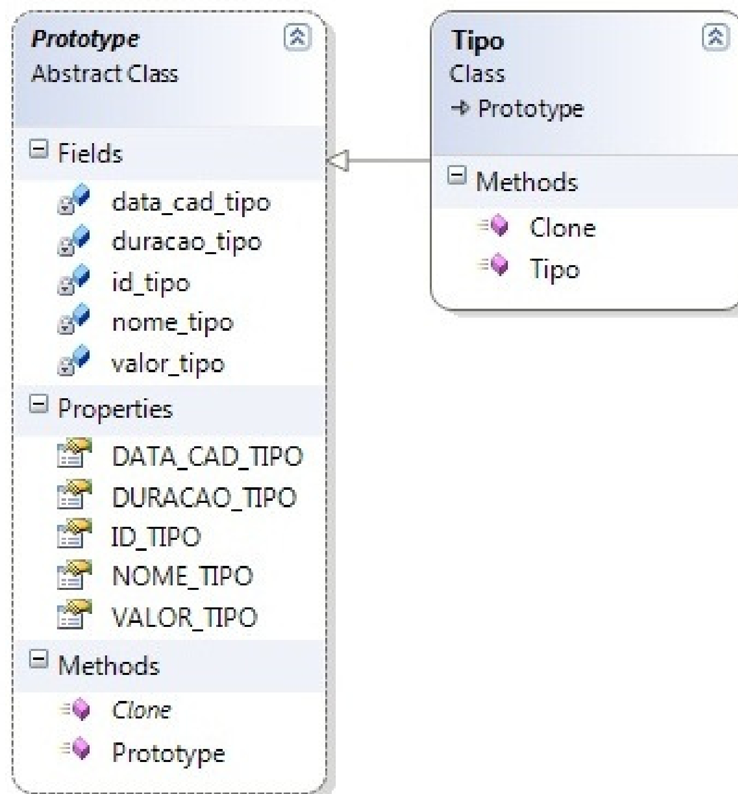

FIGURA 1 – Diagrama de classes padrão Prototype

4.3 Aplicação dos padrões estruturais

Os padrões estruturais tratam das associações entre classes e objetos.

4.3.1 Padrão decorator

O padrão Decorator adiciona responsabilidades a um objeto dinamicamente e fornece uma alternativa à criação de subclasses para estender funcionalidades.

Este padrão permite que as funcionalidades de um objeto sejam estendidas sem que seja necessário criar uma grande hierarquia de subclasses.

Para implementação do padrão Decorator foi criada uma classe abstrata Cliente com suas propriedades, atributos e com seus métodos abstratos:

```
public abstract class Cliente{
        int id_cliente;
        string nome_cliente, sobrenome_cliente, email_cliente,
senha_cliente;
        DateTime data_cad_cliente, data_nascimento_cliente;
        char sexo_cliente;
       // propriedades getters e setters
       (…)
    }
```

Criou-se a classe ClienteBasico que implementa os métodos: Inserir, Atualizar, Consulta e BuscarDadosCliente da classe abstrata Cliente. O método Inserir realiza a inserção de um cliente e suas respectivas informações no banco de dados. O método Atualizar realiza a atualização de um cliente. O método Consulta realiza uma consulta no banco de dados passando como parâmetro um inteiro identificador e retorna um objeto com as informações de um cliente. O método BuscarDadoscliente também realiza uma consulta ao banco de dados e retorna um objeto com as informações do cliente porém passa como parâmetro um DataTable. Segue o código da classe ClienteBasico (o código de implementação dos métodos foi omitido):

```
public class ClienteBasico:Cliente{
        Conexao con;
        MySqlConnection conexao = null;
        public ClienteBasico(){
            con = new Conexao();
        }
        public override bool INSERIR(clienteDTO clientedto)...
        public override bool ATUALIZAR(clienteDTO clientedto)...
        public override clienteDTO Consulta(int id)...
        public override clienteDTO BuscarDadosCliente(DataTable
dt) ...}
```

Implementou-se a classe abstrata DecoradorCliente que serve como modelo para a classe ClienteDecorado. A classe DecoradorCliente possui um atributo do tipo Cliente cujo valor é passado por parâmetro no método construtor da classe. Também possui os métodos Inserir, Atualizar, Consulta e BuscarDadosCliente, os quais também estão contidos na classe base Cliente. Para que o valor do atributo cliente,que é passado por parâmetro no método construtor da classe DecoradorCliente, seja obtido, criou-se o método getCliente.

```
  public abstract class DecoradorCliente : Cliente{
        public Cliente cliente;
        public DecoradorCliente(Cliente cliente) {
            this.cliente = cliente;
        }
        public abstract bool INSERIR(clienteDTO clientedto);
        public abstract bool ATUALIZAR(clienteDTO clientedto);
        public abstract clienteDTO Consulta(int id);
        public abstract clienteDTO BuscarDadosCliente(DataTable
dt);
        public Cliente getCliente(){
            return this.cliente;
        }
    }
```

Foi também implementada a classe ConsultaSimplificada que possui quatro atributos: nome, sobrenome, data de nascimento e sexo do cliente. O valor destes atributos são passados no método construtor da classe ConsultaSimplificada. Esta classe é utilizada apenas como tipo de retorno do método getConsultaSimplificada da classe ClienteDecorado que será mostrada mais adiante.

```
public class ConsultaSimplificada
    {
        public string nome, sobrenome;
        public DateTime DataNascimento;
        public char sexo;
        public ConsultaSimplificada(string nome, string
sobrenome, DateTime DataNascimento, char sexo)
        {
            this.nome = nome;
            this.sobrenome = sobrenome;
            this.DataNascimento = DataNascimento;
            this.sexo = sexo;
        }
    }
```

Finalmente implementou-se a classe ClienteDecorado que é classe decoradora dos objetos do tipo cliente passados como parâmetro no momento da instanciação desta classe, ou seja, a classe que adiciona novas funcionalidades ou novas regras para execução dos métodos do objeto. Segue o código da classe ClienteDecorado:

```
public class ClienteDecorado:DecoradorCliente{

        Conexao con;
        MySqlConnection conexao = null;
        public ClienteDecorado(Cliente cliente) : base(cliente)
        {
        }
        public override bool INSERIR(clienteDTO clientedto) {
            try{
                //poderia haver novas regras para modificação do
clientedto antes de realizar a inserção de fato

                this.getCliente().INSERIR(clientedto);
                return true;
            }
            catch{
                return false;
            }
            finally{
                conexao.Close();
            }
        }

        public override bool ATUALIZAR(clienteDTO clientedto)
(…)
```

```
        public override clienteDTO Consulta(int id) (…)

      public override clienteDTO BuscarDadosCliente(DataTable dt)
{
            return this.getCliente().BuscarDadosCliente(dt);
        }
        public ConsultaSimplificada getConsultaSimplificada(int
id) (...)
    }
```

Pode-se observar que classe ClienteDecorado possui a implementação dos métodos Inserir, Atualizar, Consulta e BuscarDadosCliente que já haviam sido implementados na classe ClienteBasico. Porém a implementação destes métodos na classe ClienteDecorado é diferente: antes da execução definitiva do comando no banco de dados, pode-se modificar os parâmetros passados ou pode-se adicionar novas regras para a execução do comando. Regras estas que podem ser por exemplo uma estrutura condicional: se o identificador for menor ou maior que um determinado número, executa-se o comando.

Observa-se também que a classe ClienteDecorado possui um novo método - o método getConsultaSimplificada. Método este que retorna somente o nome, o sobrenome, a data de nascimento e o sexo do cliente.

No programa principal é instanciado um objeto do tipo ClienteBasico que pode fazer uso dos métodos já implementados na classe ClienteBasico. Para que se consiga um objeto que faça uso do método getConsultaSimplificada que somente foi implementado na classe ClienteDecorado é instanciado um objeto do tipo ClienteDecorado que utiliza como parâmetro o objeto do tipo ClienteBasico já instanciado. Com isso, consegue-se adicionar uma nova funcionalidade e novas regras de execução dos comandos ao objeto do tipo ClienteBasico.

Segue o exemplo do código utilizado na página para adicionar uma nova funcionalidade ao objeto:

```
            //Instância do objeto Cliente.
            Cliente cliente = new ClienteBasico();
            cliente.ID_CLIENTE = id;
            //Instância do obejto decorator. Aqui é passado o
objeto a ser decorado
            ClienteDecorado decoratorCliente = new
ClienteDecorado(cliente);
```

```
            ConsultaSimplificada consultasimples =
decoratorCliente.getConsultaSimplificada(id);
```

Na FIG. 2 está ilustrado o diagrama das classes utilizadas para implementação do padrão Decorator. Estão ilustradas as classes abstratas Cliente e DecoradorCliente com seus respectivos atributos, métodos e propriedades. A figura também mostra as classes ClienteBasico, ClienteDecorado e ConsultaSimplificada com seus respectivos atributos e métodos.

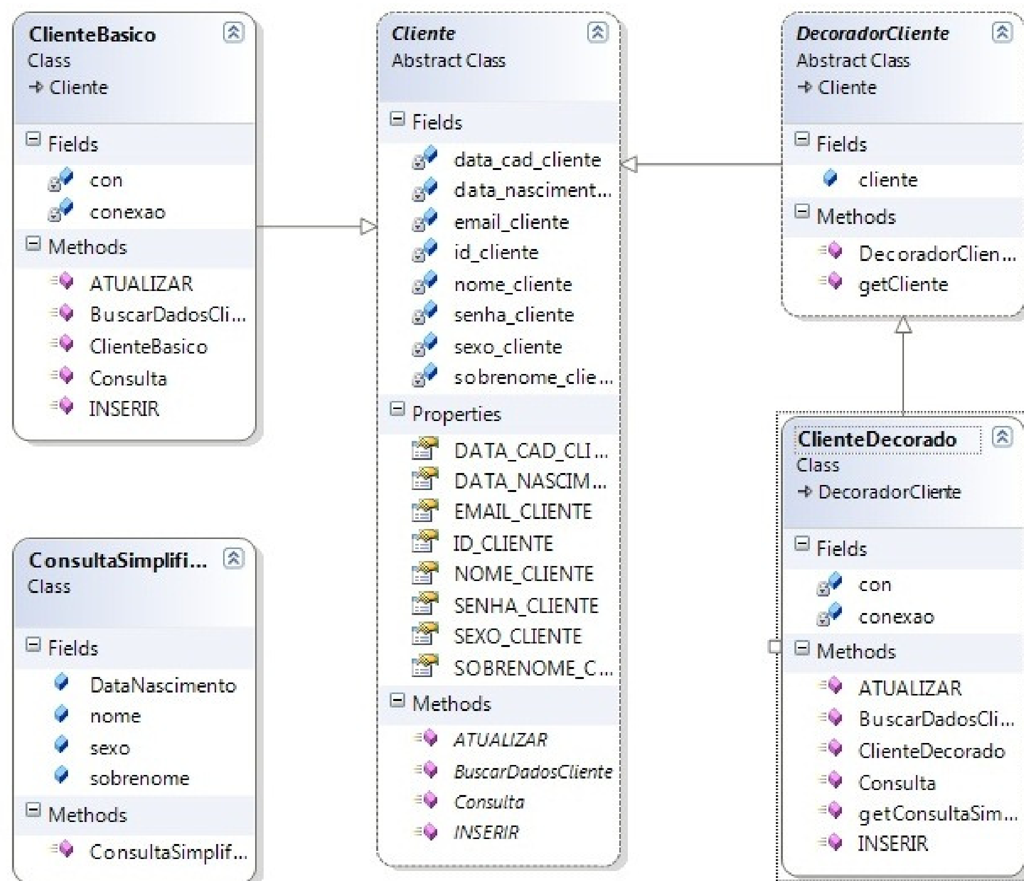

FIGURA 2 – Diagrama de classes padrão Decorator

4.4 Aplicação dos padrões comportamentais

Os padrões comportamentais tratam das interações e divisões de responsabilidades entre as classes ou objetos.

4.4.1 Padrão DTO(data transfer object)

Para implementação do padrão DTO foi utilizada a arquitetura em camadas e foi criada a camada DTO que contém as classes que encapsulam os atributos e propriedades.

A camada DTO possui as classes cartaoDTO, categoriaDTO, clienteDTO, filmeDTO, tipoDTO e vendaDTO. A classe cartaoDTO é utilizada para armazenar informações dos cartões de um cliente. A classe categoriaDTO é usada para armazenar informações a respeito da categoria de um filme. A classe clienteDTO é utilizada para armazenar informações de um cliente. Objetos do tipo filmeDTO armazenam informações de filmes. Informações a respeito do tipo dos filmes são armazenadas em objetos do tipo tipoDTO. A classe vendaDTO é usada para armazenar informações a respeito das vendas.

Esta implementação possibilita um ganho na transferência de dados já que é passado somente um objeto e não vários parâmetros em invocações de métodos e consultas.

Sem a utilização do DTO os parâmetros são passados um a um no momento da invocação dos métodos:

```
public bool INSERIR(string nome_cliente,string sobrenome_cliente, string email_cliente, string senha_cliente, datetime data_cad_cliente, datetime data_nascimento_cliente, char sexo_cliente) ...
```

Fazendo uso do padrão DTO é passado somente o objeto de transferência como parâmetro na invocação dos métodos:

```
public bool INSERIR(clienteDTO clientedto)...
```

A FIG. 3 mostra o diagrama das classes utilizadas para implementação do padrão DTO. Podem ser observadas as classes cartaoDTO, clienteDTO, tipoDTO, categoriaDTO, filmeDTO, vendaDTO e os seus respectivos atributos e métodos.

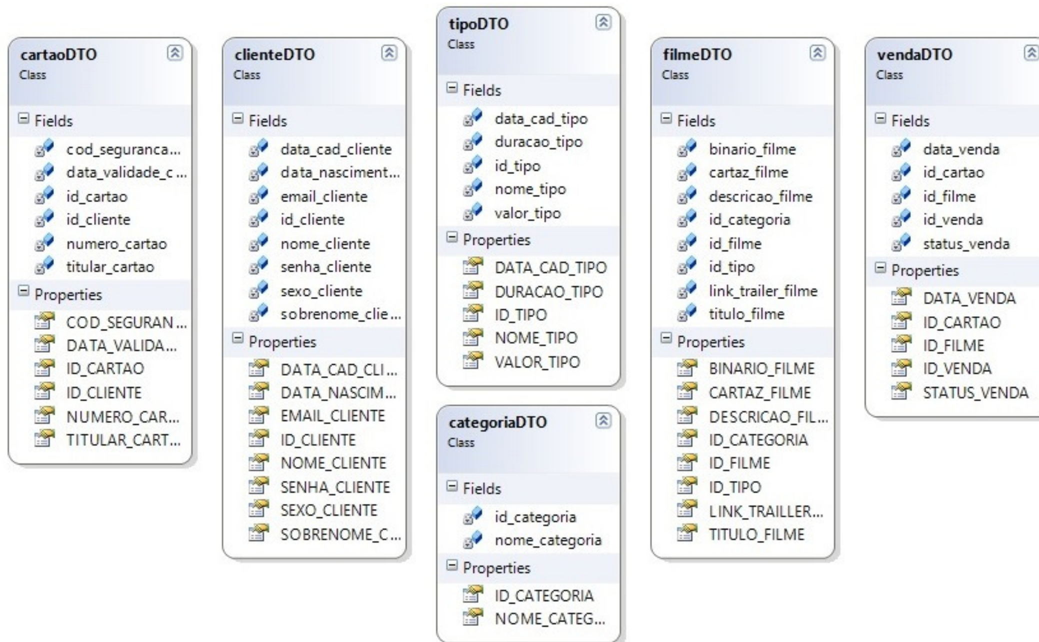

FIGURA 3 – Diagrama de classes padrão DTO

5. RESULTADOS E DISCUSSÃO

No desenvolvimento da segunda versão do sistema Netmovies com a refatoração do código pode-se observar resultados específicos conseguidos com a aplicação de cada padrão de projeto.

Com a aplicação do padrão Factory Method, pode-se notar um maior controle sobre a instância de objetos de acordo com tipo desejado. A classe ConexaoFactory se tornou a responsável por decidir qual instância deveria retornar: decide se retorna um objeto de conexão local ou um objeto de conexão remota.

O uso do padrão Prototype permitiu a definição da classe Prototype, que foi usada para que objetos deste tipo de classe fossem clonados. Este fato possibilita um trabalho mais fácil quando se necessita de objetos deste tipo de classe.

O padrão Singleton permitiu o controle da instância da classe FilmeSingleton: apenas uma classe deste tipo foi permitida. Quando se requisita uma instância da classe FilmeSingleton é verificado se já existe uma

instância desta classe. Se já houver uma instância desta classe, é retornada a instância já existente. Senão, instancia-se a classe e então retorna a sua instância.

O uso do padrão Adapter permitiu que a nova classe CartaoAdapter, otimizada e com nomes de métodos diferentes, fosse compatível com uma classe diferente anteriormente definida: a classe cartaoDAL.

O padrão Decorator fez com que novas responsabilidades e novas regras de execução fossem adicionadas a um objeto. Isto é possível com a implementação da classe ClienteDecorado que recebe um objeto como parâmetro em seu método construtor e adiciona as responsabilidades e novas regras de execução a este objeto.

A aplicação do padrão Command tornou possível o encapsulamento de uma requisição como um objeto. Tornou possível também que se separasse a classe que contém o código de execução do comando da classe que realiza a chamada ao comando.

O padrão DTO possibilitou um ganho na transferência de dados – em vez de se passar vários parâmetros na chamada a um método passa-se apenas um objeto.

O uso do padrão Iterator permitiu o acesso aos elementos de um objeto sequencialmente sem que fosse necessário conhecer a sua estrutura interna. Possibilitou o acesso mais fácil aos elementos de uma lista.

A aplicação do padrão Memento fez com que o estado de um objeto fosse capturado para que pudesse ser restaurado posteriormente.

O padrão Strategy permitiu que uma família de algoritmos fosse encapsulada e permitiu a variação destes algoritmos independentemente das classes clientes que faziam uso destes algoritmos.

6. CONCLUSÃO

Pode-se concluir que com a aplicação dos padrões é possível obter componentes reutilizáveis. O trabalho permitiu a demonstração da aplicação dos padrões de projeto.