Instituto Federal de Educação, Ciência e
Tecnologia de São Paulo
Câmpus Birigui

ALINE BERTOLAZO DOS SANTOS
ANDRE LUIZ DA SILVA CONDE
ANDREY MATHEUS BRAMBILLA
ISADORA D BUENO DOS SANTOS
VINICIUS DE SOUZA SANTOS

Padrão de Projeto

Análise de Projeto e Sistemas

Birigui

2022

Sumário

1 Introdução

Você não precisa ter uma vasta experiência em desenvolvimento de software orientado a objetos para ter ouvido falar ou usado padrões de projeto. Projetos de software de diferentes domínios e tamanhos utilizam essa técnica e, ao longo dos anos, essa técnica mostrou seu valor, proporcionando bons resultados, principalmente em termos de reutilização de código, economizando tempo e dinheiro. Pode ser que você já conheça e tenha aplicado esses padrões no seu projeto, ou você sabe, mas ainda não aplica, pode ser que você não conheça e não aplique, mas talvez lendo isso você identifique projetos onde os padrões de projeto podem fornecer soluções importantes em vários pontos.

Os padrões de design surgiram na década de 1970 para ajudar a resolver problemas recorrentes, inclusive em projetos de desenvolvimento de software. Na superfície, são soluções abrangentes para problemas conhecidos, pelo menos até que surja outra solução melhor. Eles são o resultado da experiência de muitos desenvolvedores, trabalho duro e muitas tentativas e erros, e representam as melhores práticas em programação orientada a objetos.

Na arquitetura de software, o mais famoso conjunto de padrões originou-se no livro dos autores Gamma, Helm, Johnson e Vlissides, mais conhecido como Gang of Four ou GoF: Design Patterns: Elements of Reusable Objected - Software oriented. Este trabalho é continuamente consultado e ajuda diretamente os desenvolvedores de todo o mundo a aplicar o padrão em seus projetos. Os autores identificaram 23 padrões e os dividiram em 3 grupos: padrões de nutrição, padrões estruturais e padrões comportamentais.

2 Facade

2.1 Definição

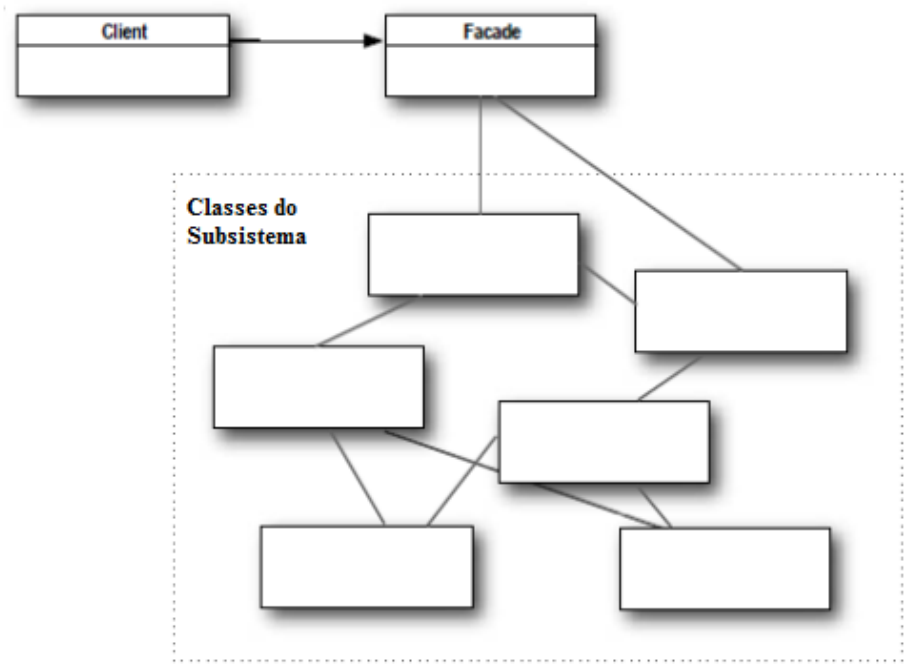
O Padrão de Projeto Facade oculta toda a complexidade de uma ou mais classes através de uma Facade (Fachada). A intenção desse Padrão de Projeto é simplificar uma interface. Existem outros dois Padrões de Projetos (Decorator e Adapter) já discutidos em outros artigos que possuem similaridades com o Padrão Facade, porém existem diferenças em relação a este padrão, como será visto mais adiante. O Padrao Facade e um um padrão de projeto estrutural, estudiosos afirma que esse e o padrão de projeto mais simples de ser implentado.

Com o Padrão Facade podemos simplificar a utilização de um subsistema complexo apenas implementando uma classe que fornece uma interface única e mais razoável, porém se desejássemos acessar as funcionalidades de baixo nível do sistema isso seria perfeitamente possível. É importante ressaltar que o padrão Facade não “encapsula” as interfaces do sistema, o padrão Facade apenas fornece uma interface simplificada para acessar as suas funcionalidades. Imagine que existe um sistema com diversas classes contendo diversos métodos e tenhamos que agrupar todas essas classes chamando diversos métodos para realizar uma determinada operação. Tendo uma Facade precisaríamos apenas construir um método que agrupe todas essas classes e chame todos esses métodos. Assim, quando usuário quiser fazer essa operação ele chamaria

apenas a Facade que realizaria essa operação, simplificando muito todo o processo com uma simples interface. Vale ressaltar que isso não significa que uma Facade não tenha também funcionalidades próprias, ou seja, que tenha a sua própria inteligência e também utilize o subsistema. Um subsistema pode ter diversos Facades.

O Diagrama de classe abaixo mostra mais detalhes sobre o funcionamento do padrão Facade.

Figure 1 Diagrama de classe do Padrão Facade



No diagrama de classe acima tem-se o Client que é quem acessa a Facade que, por sua vez, é uma interface simplificada do subsistema, sendo esta unificada e fácil de ser utilizada pelo cliente. Abaixo da Facade tem-se as classes do sistema que podem ser chamados diretamente, mas estão sendo agrupados na Facade.

Em outras maneiras de dizer, o Facade vem do inglês que significa fachada, ou seja, e uma fachada do sistema, para um software que possui uma estruturação mais complexa, s função desse Padrão e tornar a complexidade do sistema oculta e melhorar a funcionalidade do Sistema, porem temos que tomar cuidado para ela não se tornar uma God Class “Classe faz tudo”, pois quebra o conceito de princípio de responsabilidade única se isso ocorrer podemos criar outras classe de fachada.

2.2 Implementação na Prática

Segue abaixo um exemplo de implementação em Java utilizando o Padrão Facade.

```
public class Cpu {

   public void start() {
      System.out.println("inicialização inicial");
   }
   public void execute() {
      System.out.println("executa algo no processador");
   }
   public void load() {
      System.out.println("carrega registrador");
   }
   public void free() {
      System.out.println("libera registradores");
   }
}

public class Memoria {
   public void load(int position, String info) {
      System.out.println("carrega dados na memória");
   }
   public void free(int position, String info) {
      System.out.println("libera dados da memória");
   }
}

public class HardDrive {
   public void read(int startPosition, int size) {
      System.out.println("lê dados do HD");
   }
   public void write(int position, String info) {
      System.out.println("escreve dados no HD");
   }
}

public class ComputadorFacade {
```

```
    private Cpu cpu = null;
    private Memoria memoria = null;
    private HardDrive hardDrive = null;

    public ComputadorFacade(Cpu cpu,
                Memoria memoria,
                HardDrive hardDrive) {
        this.cpu = cpu;
        this.memoria = memoria;
        this.hardDrive = hardDrive;
    }

    public void ligarComputador() {
        cpu.start();
        String hdBootInfo = hardDrive.read(BOOT_SECTOR, SECTOR_SIZE);
        memoria.load(BOOT_ADDRESS, hdBootInfo);
        cpu.execute();
        memoria.free(BOOT_ADDRESS, hdBootInfo);
    }
}
```

No exemplo acima podemos notar a quantidade de classes e métodos envolvidos quando precisamos inicializar o computador. Toda essa complexidade é exposta ao cliente que poderia chamar todas essas classes e cada um dos métodos das classes para realizar a tarefa de inicializar o computador. No entanto, ao usar uma Facade encapsulamos essa complexidade oferecendo uma interface simples e unificada ao cliente evitando acoplamento e complexidade. Apenas chamando o método ligarComputador() da classe ComputadorFacade tem-se uma interface simples que diz o que ela faz exatamente, sem expor a complexidade envolvida na operação.

Nota-se que todas as chamadas que estão no Facade poderiam ser feitas uma a uma no cliente, porém isso gera muito acoplamento e complexidade para o cliente, por isso a Facade simplifica e unifica esse conjunto de classes que gera muita complexidade.

2.3 Conclusão

O Padrão Facade é utilizado quando precisamos simplificar e unificar uma interface grande ou um conjunto complexo de interfaces. Uma das vantagens do padrão Facade é desconectar o cliente de um subsistema complexo, conforme pode ser visto no diagrama de classes. Um sistema pode ter diversos Facades simplificando diversos pontos do programa.

3 Flyweight

3.1 Definição

O Padrão Flyweight vem do inglês e seu significado e mosca, esse padrão de projeto estrutural carrega em si o conceito de otimizar seu código, ou seja, reduzir espaço quando vários objetos precisam ser manipulados na memória, Ao criarmos muitos objetos idênticos, o Flyweight pode diminuir a quantidade de memória que está sendo usada para um nível, de certa forma, administrável.

A principal intenção deste padrão é estruturar objetos de modo que eles possam ser compartilhados entre diversos contextos. Ele é, muitas vezes, confundido como sendo uma “fábrica”, que é responsável pela criação do objeto. A estrutura do padrão envolve uma fábrica Flyweight para criar a correta implementação da interface do Flyweight, mas com certeza não são os mesmos padrões utilizados.

A programação orientada a objeto é considerada por muitos desenvolvedores uma bênção, no entanto, ela tem alguns desafios. Considere então um grande objeto de domínio, no qual tenhamos que modelar cada componente como sendo um objeto. Às vezes, programas trabalham com muitos objetos que têm a mesma estrutura e alguns estados desses objetos não variam no tempo. Quando usamos uma abordagem tradicional e criamos instâncias desses objetos e preenchemos variáveis de estado com valores, os requisitos de memória e armazenamento podem inaceitavelmente aumentar. Para resolvermos este tipo de situação, podemos então usar o padrão Flyweight.

O padrão Flyweight pode vir para nós auxiliares, pois ele ajuda a reduzir o custo de armazenamento de muitos objetos. Ele também nos permite partilhar objetos em vários contextos simultaneamente. O padrão permite alcançar esses objetivos por retenção orientada a objetos de granularidade e flexibilidade ao mesmo tempo.

A solução ideal para situações em que a criação de objetos compartilhados seja necessária pode ser oferecida pelo Flyweight, onde a chave para criarmos os objetos compartilhados é a distinção entre o estado intrínseco e extrínseco de um objeto. Os objetos compartilhados no padrão são chamados Flyweights. Tratando um pouco sobre o estado extrínseco, ele é fornecido como um contrapeso a partir do exterior como um parâmetro, quando alguma operação é chamada nele. Esse estado não é armazenado dentro do Flyweight.

A solução implementada pelo padrão Flyweight é bem intuitiva. No entanto vale a pena comentar alguns detalhes. Percebeu que, na classe fábrica fica centralizado o acesso a todos os objetos compartilhados? O aconteceria se houvessem duas ou mais instâncias desta classe? Seriam criados vários objetos, sem nenhuma necessidade.

Para evitar este problema vale a pena dar uma olhada em outro padrão, o Singleton. Aplicando este padrão na classe fábrica, garantimos que apenas uma instância dela será utilizada em todo o projeto.

O ponto fraco do padrão é que, dependendo da quantidade e da organização dos objetos a serem compartilhados, pode haver um grande custo para procura dos objetos compartilhados. Então ao utilizar o padrão deve ser analisado qual a prioridade no projeto, espaço ou tempo de execução.

Imagine que existe um grupo de objetos que serão compartilhados juntos, por exemplo, uma sequência de objetos do cenário. Nesta situação, existe uma combinação com outro padrão, o Composite. Com ele é possível agrupar um conjuntos de objetos Flyweight que serão compartilhados juntos.

Outro ponto de interesse é a instanciação de todos os objetos flyweight na classe fábrica. Suponha que algum objeto é instanciado, mas nunca é utilizado? Pode ser implementado uma estratégia de garbage collection, que controla o número de instâncias de um determinado objeto. Ao não ser mais utilizado, o objeto é liberado da memória, reduzindo mais ainda o espaço.

3.2 Na Prática

No desenvolvimento de jogos são utilizadas várias imagens. Elas representam as entidades que compõe o jogo, por exemplo, cenários, jogadores, inimigos, entre outros. Ao criar classes que representam estas entidades, é necessário vincular a elas um conjunto de imagens, que representam as animações.

Quem desenvolve jogos pode ter pensado na duplicação de informação quando as imagens são criadas pelos objetos que representam estas entidades, por exemplo, a classe que representa um inimigo carrega suas imagens. Quando são exibidos vários inimigos do mesmo tipo na tela, o mesmo conjunto de imagens é criado repetidamente.

A solução para esta situação de duplicação de informações pelos objetos é a utilização do padrão Flyweight.

A intenção do padrão:

“Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina.” [1]

Pela intenção percebemos que o padrão Flyweight cria uma estrutura de compartilhamento de objetos pequenos. Para o exemplo citado, o padrão será utilizado no compartilhamento de imagens entre as entidades.

Antes de exemplificar vamos entender um pouco sobre a estrutura do padrão. A classe Flyweight fornece uma interface com uma operação que deve ser realizado sobre um estado interno. No exemplo esta classe irá fornecer uma operação para desenhar a imagem em um determinado ponto. Desta forma a imagem é o estado intrínseco, que consiste em uma informação que não depende de um contexto externo. O ponto passado como parâmetro é o estado extrínseco, que varia de acordo com o contexto.

Vamos então ao código da imagem e do ponto:

```
public class Imagem {
    protected String nomeDaImagem;

    public Imagem(String imagem) {
        nomeDaImagem = imagem;
    }

    public void desenharImagem() {
        System.out.println(nomeDaImagem + " desenhada!");
    }
}
```

Para simplificar o exemplo, será apenas exibida uma mensagem no terminal, indicando que a imagem foi desenhada.

```
public class Ponto {
    public int x, y;

    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

A classe Flyweight vai apenas fornecer a interface para desenho da imagem em um ponto.

```
public abstract class SpriteFlyweight {
    public abstract void desenharImagem(Ponto ponto);
}
```

Outro componente da estrutura do Flyweight é a classe Flyweight concreta, que implementa a operação de faot:

```
public class Sprite extends SpriteFlyweight {
    protected Imagem imagem;

    public Sprite(String nomeDaImagem) {
        imagem = new Imagem(nomeDaImagem);
    }

    @Override
    public void desenharImagem(Ponto ponto) {
        imagem.desenharImagem();
        System.out.println("No ponto (" + ponto.x + "," + ponto.y + ")!");
    }
}
```

Nesta classe também será apenas exibida uma mensagem no terminal para dizer que a imagem foi desenhada no ponto dado. O próximo componente da estrutura do Flyweight

consiste em uma classe fábrica, que vai criar os vários objetos flyweight que serão compartilhados.

```
1
2
3public class FlyweightFactory {
4
5    protected ArrayList<SpriteFlyweight> flyweights;
6
7    public enum Sprites {
8        JOGADOR, INIMIGO_1, INIMIGO_2, INIMIGO_3, CENARIO_1, CENARIO_2
9    }
10
11   public FlyweightFactory() {
12       flyweights = new ArrayList<SpriteFlyweight>();
13       flyweights.add(new Sprite("jogador.png"));
14       flyweights.add(new Sprite("inimigo1.png"));
15       flyweights.add(new Sprite("inimigo2.png"));
16       flyweights.add(new Sprite("inimigo3.png"));
17       flyweights.add(new Sprite("cenario1.png"));
18       flyweights.add(new Sprite("cenario2.png"));
19   }
20
21   public SpriteFlyweight getFlyweight(Sprites jogador) {
22       switch (jogador) {
23       case JOGADOR:
24           return flyweights.get(0);
25       case INIMIGO_1:
26           return flyweights.get(1);
27       case INIMIGO_2:
28           return flyweights.get(2);
29       case INIMIGO_3:
30           return flyweights.get(3);
31       case CENARIO_1:
32           return flyweights.get(4);
33       default:
34           return flyweights.get(5);
35       }
   }
}
```

Além de criar os vários objetos a serem compartilhados, a classe fábrica oferece um método para obter o objeto, assim, o acesso a estes objetos fica centralizado e unificado a partir desta classe.

Para exemplificar a utilização do padrão, vejamos o seguinte código cliente:

```
1public static void main(String[] args) {
2    FlyweightFactory factory = new FlyweightFactory();
3
4    factory.getFlyweight(Sprites.CENARIO_1).desenharImagem(new Ponto(0, 0));
5
```

```
6       factory.getFlyweight(Sprites.JOGADOR).desenharImagem(new Ponto(10, 10));
7
8       factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(
9               new Ponto(100, 10));
10      factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(
11              new Ponto(120, 10));
12      factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(
13              new Ponto(140, 10));
14
15      factory.getFlyweight(Sprites.INIMIGO_2).desenharImagem(
16              new Ponto(60, 10));
17      factory.getFlyweight(Sprites.INIMIGO_2).desenharImagem(
18              new Ponto(50, 10));
19
20      factory.getFlyweight(Sprites.INIMIGO_3).desenharImagem(
21              new Ponto(170, 10));
22  }
```

É exibido um conjunto de imagens para exemplificar o uso em um jogo. São desenhados inimigos de vários tipos, o cenário do jogo e o jogador. Note que o acesso aos objetos fica centralizado apenas na classe fábrica.

No desenvolvimento de jogos real, as referências dos objetos seriam espalhadas pelas entidades, garantindo a não duplicação de conteúdo.

4 Proxy

4.1 Definição

O padrão Proxy tem como objetivo proporcionar um espaço reservado para outro objeto controlar o acesso a ele. A classe proxy teoricamente pode se conectar a qualquer objeto, ou seja, normalmente quando existe uma instância grande/complexa pode-se criar vários proxies, todos contendo uma única referência.

4.1.1 Classificação

Padrão de projeto descreve 23 modelos de desenho, porém cabe ressaltar que existem centenas de padrões. No famoso livro Design Patterns dos autores Gamma, Helm, Johnson e Vlissides é apresentado 23 padrões de projeto, documentados e já conhecidos no mundo do desenvolvimento de software. É importante saber que isso não significa que esses padrões são os mais uteis para se implementar, sendo assim vale a pena pesquisar padrões de outras fontes.

Esses 23 padrões são divididos em padrões de criação, padrões estruturais e padrões comportamentais. Vamos explorar os 23 padrões de projeto na série de artigo sobre o assunto. O padrão de projeto Proxy está classificado como padrão estrutural.

4.2. Implementando padrão Proxy

Geralmente os objetos comuns fazem seu próprio trabalho e oferecem suporte as interfaces públicas declaradas no objeto. Porém existem casos que essa responsabilidade precisa ser transferida para um objeto adequado. Isto pode ocorrer em diversos casos, vamos para alguns exemplos: quando um objeto necessita de um longo tempo para carregar; quando o programador precisa intercepatar a mensagem do objeto; quando o processo precisa ser executado em outro computador/servidor e etc. Nestes casos, um objeto proxy pode assumir a responsabilidade, assim criando uma instância cliente que aguarda e encaminha solicitações de forma adequada a um objeto alvo.

Com isso podemos entender o principal objetivo do padrão Proxy, que em resumo é fornecer uma solução substituta ou espaço reservador para outro objeto controlar a solicitação.

4.2.1 Simple Proxy

Um objeto proxy geralmente possui uma interface que é praticamente idêntica à interface do proxy que irá substituí-lo. O proxy vai realizar seu trabalho criteriosamente por encaminhamento de solicitações para o objeto subjacente. Um exemplo prático do proxy refere-se a evitar a utilização inadequada do espaço em memória. Suponhamos que arquivos/dados em um aplicativo pertencem a tarefas que inicialmente não serão executadas, sendo assim, precisamos evitar o carregamento desnecessário de todos os arquivos/dados antes que eles sejam solicitados, neste caso pode deixar os proxies para os arquivos. Essa atividade funciona como um espaço reservado para carregar os arquivos requeridos na demanda, vamos desenvolver um exemplo simples que ilustra essa situação.

Abaixo o código da classe principal, fazendo a chamada a classe que carrega o arquivo solicitado.

```
class Program

{

    static void Main(string[] args)

    {

        Arquivo arq01 = new ArquivoProxy("Arquivo01");

        Arquivo arq02 = new ArquivoProxy("Arquivo02");

        Arquivo arq03 = new ArquivoProxy("Arquivo03");



        Console.ReadKey();

    }
```

```
    }
```

Interface para implementação do padrão proxy.

```
    interface Arquivo
    {
        void getArquivo();
    }
```

Classe que contém o método que será executado pelo proxy virtual.

```
    public class ArquivoReal:Arquivo
    {
        private string NomeDoArquivo;
        public ArquivoReal(string _nomedoarquivo)
        {
            this.NomeDoArquivo = _nomedoarquivo;
            loadArquivo();
        }
        private void loadArquivo()
        {
            Console.WriteLine("Carregando: " + this.NomeDoArquivo);
        }
        public void getArquivo()
        {
            Console.WriteLine(this.NomeDoArquivo);
        }
   }
```

Classe que ilustra uma forma simples de proxy virtual. A classe abaixo será utilizada para acessar um método remoto.

```
    public class ArquivoProxy : Arquivo {
        private string NomeDoArquivo;
```

```
        private Arquivo vrArquivo;

        public ArquivoProxy(string _nomedoarquivo)

        {

            this.NomeDoArquivo = _nomedoarquivo;

            getArquivo();

        }

        public void getArquivo()

        {

            vrArquivo = new ArquivoReal(this.NomeDoArquivo);

            vrArquivo.getArquivo();

        }

    }
```

Abaixo o resultado do programa:

Carregando: Arquivo01

Arquivo01

Carregando: Arquivo02

Arquivo02

Carregando: Arquivo03

Arquivo03

Este exemplo ilustra de uma forma simples a implementação do pradrão proxy, porem ainda não deixa claro a total justificatica para utilizar esse padrão. No próximo exemplo vamos desenvolver um proxy com chamada remota, justificando os beneficios em se usar o padrão proxy.

4.2.2 Remote Proxies

Em algumas situações precisamos chamar um método que está sendo executado em outro computador, neste caso é interessante encontrar uma maneira de se comunicar com o objeto remoto sem chamá-lo diretamente. Uma forma de executar essa atividade seria abrir um socket no servidor remoto e elaborar algum protocolo para transferir mensagens entre os objetos. Este processo permite passar mensagens para os objetos remotos como fossem hospedados no servidor local.

Existem objetos bem conhecidos que é capaz de chamar métodos que estão sendo executados em máquinas remotas. Alguns componentes são: Common Object Request Broker Architecture (CORBA), Java Remote Method (RMI), Active Server Pages em ASP.NET.

4.2.3 Exemplo

Mão na massa, este exemplo requer os seguintes passos:

- Internet Information Server (IIS)
- Relacionar um diretório no disco com um endereço Uniform Resource Locator (UML)
- Web services (vamos desenvolver juntos)
- Criar um proxy para o serviço de web (vamos criar juntos)

Passo 1**:** Instalar o IIS, caso a ferramenta ainda não esteja disponível.

Passo 2: Criar um diretório como por exemplo "c:\exemplo" e o diretório onde será hospedado nossa pagina como por exemplo "c:\exemplo\bin". Execute o IIS e crie um diretório virtual apontando para pasta "c:\exemplo" e informe o *alias name* "exemploproxy".

Passo 3: Vamos criar um Web service

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Services;



[WebService(Namespace = "http://tempuri.org/")]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]

public class WebService : System.Web.Services.WebService {

    public WebService () {

    }

    [WebMethod]

    public string OlaMundo() {

        return "Web service. Olá mundo";
```

```
    }

}
```

Passo 5: Capturar a interface do serviço de Web service disponível. Com o Web Services Description Language (WSDL), este recurso faz parte do conteúdo disponível no kit de ferramentas do .NET framework. Execute o seguinte comando:

c:\exemplo>wsdl **http://localhost/exemplo/WebService.asmx**?wsdl

Após executar esse comando o arquivo Webservice.cs seré criado contendo o proxy que os clientes porem utilizar para chamar o serviço. Podemos traduzir essa processo da seguinte forma. O arquivo criado irá conter uma chamada ao método OlaMundo() que vai operar de forma diferente da OlaMundo() implementada no WebService.asmx. Quando o método for solicitado será criado um pedido no formato texto correspondente ao protocolo Simples Object Access Protocol (SOAP), passando essa mensagem para o servidor IIS que envia a resposta para o cliente que está aguardando.

Passo 6: No ultimo passo, basta então criar o cliente que irá utilizar o proxy. Crie um novo projeto, adicione a ele a classe WebService.cs (criada com o comando wsdl) e implemente o código abaixo no método Main().

```
class MostraCliente

{

    static void Main()

    {

        // Metodo disponível no arquivo webservice.cs,

        //criado pelo comando wsdl

         OlaMundo _olamundo =

             new WebService.OlaMundo();

    }

}
```

Um dos benefícios do ASP.NET é que a framework permite que aplicativo cliente possa interagir com um objeto local, no caso o proxy para o objeto remoto. Com isso os desenvolvedores dificilmente precisam estar cientes das chamadas remotas que acontecem, sendo que o ASP.NET fornece um meio de comunicação e isola o servidor do cliente.

4.3 Conclusão

O proxy na computação distribuída é um avanço permanente em computação orientada a objeto. É recomendado então implementar o padrão proxy, quando existe a necessidade de estabelecer um objeto reservado que controla/gerencia o acesso a outra tarefa. O objeto proxy tem a habilidade de isolar o cliente a partir de uma mudança no estado do objeto, como vimos nos exemplos acima.

O padrão proxy tem alguns problemas, podemos perceber que o objeto depende de uma ligação entre o espaço reservado e o objeto em proxy. Geralmente na prática, o uso de proxy pode trazer benefícios, porém os projetos alternativos muitas vezes oferecem outras soluções mais estáveis. O uso de proxy é recomendado para computação remota, onde realmente justifica o uso dele. Ao invés de depender de algum outro protocolo ou sistema de computação distribuída. Como o ASP.NET estabelece comunicação com objetos remotos em uma chamada de método normal, este recurso permite que o cliente se comunique com outro objeto utilizando o proxy como referência, caracterizando como se o acesso fosse local.

Referencias

Uma introdução aos Padrões de Projeto. Disponível em: <https://www.igti.com.br/uma-introducao-aos-padroes-de-projeto>. Acesso em: 22 maio. 2022.

MEDEIROS, H. **Padrão de Projeto Facade em Java**. Disponível em: <https://www.devmedia.com.br/padrao-de-projeto-facade-em-java/26476>. Acesso em: 22 maio. 2022.

Mão na massa: Flyweight. Disponível em: <https://brizeno.wordpress.com/2011/11/13/mao-na-massa-flyweight/>. Acesso em: 22 maio. 2022.

FLAVIO SECCHIERI MARIOTTI**. Padrão de Projeto Proxy em .NET**. Disponível em: <https://www.devmedia.com.br/padrao-de-projeto-proxy-em-net/22183>. Acesso em: 22 maio. 2022.