Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

outubro 3, 2011

Mão na massa: Adapter

O padrão Adapter:

Problema

A utilização de frameworks é muito comum atualmente, devido a comodidade e facilidade de utilização. No entanto discute-se muito, principalmente na comunidade ágil, sobre a postura dos programadores frente aos frameworks.

O ponto de discussão é que a utilização de frameworks deve ser considerada algo fora do projeto original, e não um componente interno que torna todo o sistema dependente de um framework específico.

Vamos considerar o seguinte exemplo: é preciso fazer um sistema que manipule imagens, para isto será utilizado uma API que oferece essas funcionalidades. Suponhamos que será necessário ter um método para carregar a imagem de um arquivo e outro para exibir a imagem na tela.

Como podemos então construir o sistema de maneira que ele fique independente de qual API será utilizada?

Para exemplificar considere as duas classes a seguir que representam as APIs utilizadas. Como estas classes fazem parte da API não temos condições de alterá-las.

```
1  public class OpenGLImage {
2
3      public void glCarregarImagem(String arquivo) {
4          System.out.println("Imagem " + arquivo + " carregada.");
5      }
6
7      public void glDesenharImagem(int posicaoX, int posicaoY) {
8          System.out.println("OpenGL Image desenhada");
9      }
10 }
```

```
1  public class SDL_Surface {
2
3      public void SDL_CarregarSurface(String arquivo) {
4          System.out.println("Imagem " + arquivo + " carregada.");
5      }
6
7      public void SDL_DesenharSurface(int largura, int altura, int posicaoX,
8              int posicaoY) {
9          System.out.println("SDL_Surface desenhada");
10     }
11
12 }
```

Como podemos então, além de deixar o sistema independente, unificar uma interface de acesso para qualquer API? Veja que a assinatura dos métodos é bem diferente, desde o nome até quantidade de parâmetros.

Poderíamos utilizar o Template Method para definir uma interface e deixar cada subclasse redefinir algumas operações, no entanto não existe aqui a ideia de um "algoritmo comum".

Vejamos agora o que é o padrão Adapter.

Adapter

Intenção:

"Converter a interface de uma classe em outra interface, esperada pelo cliente. O Adapter permite que interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível." [1]

Ou seja, dado um conjunto de classes com mesma responsabilidade, mas interfaces diferentes, utilizamos o Adapter para unificar o acesso a qualquer uma delas.

Precisamos então, inicialmente, fornecer uma interface comum para o cliente, oferencendo o comportamento que ele necessita:

```
1  public interface ImagemTarget {
2      void carregarImagem(String nomeDoArquivo);
3
4      void desenharImagem(int posX, int posY, int largura, int altura);
5  }
```

Agora vamos definir os adaptadores:

```java
public class OpenGLImageAdapter extends OpenGLImage implements ImagemTarg

    @Override
    public void carregarImagem(String nomeDoArquivo) {
        glCarregarImagem(nomeDoArquivo);
    }

    @Override
    public void desenharImagem(int posX, int posY, int largura, int altur
        glDesenharImagem(posX, posY);
    }

}
```

```java
public class SDLImagemAdapter extends SDL_Surface implements ImagemTarget

    @Override
    public void carregarImagem(String nomeDoArquivo) {
        SDL_CarregarSurface(nomeDoArquivo);
    }

    @Override
    public void desenharImagem(int posX, int posY, int largura, int altur
        SDL_DesenharSurface(largura, altura, posX, posY);
    }
}
```

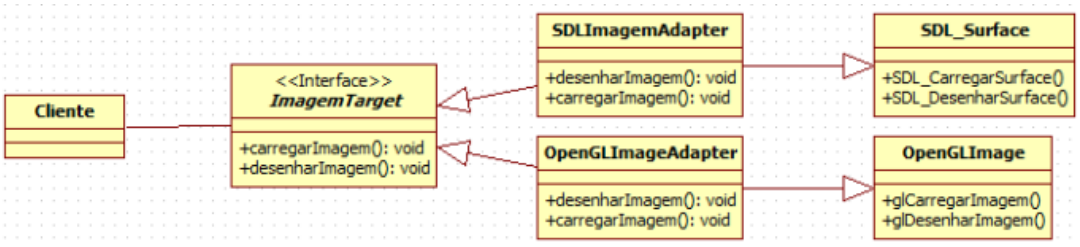
De posse dos adaptadores, nosso cliente fica então independente de qual API será utilizada, utilizando apenas a Interface comum definida no início:

```java
public static void main(String[] args) {
    ImagemTarget imagem = new SDLImagemAdapter();
    imagem.carregarImagem("teste.png");
    imagem.desenharImagem(0, 0, 10, 10);

    imagem = new OpenGLImageAdapter();
    imagem.carregarImagem("teste.png");
    imagem.desenharImagem(0, 0, 10, 10);
}
```

Qualquer mudança em qual API será utilizada é facilmente feita, sem a necessidade de alterar o programa inteiro. Caso fosse necessário utilizar uma nova API também seria simples, bastaria criar o adaptador para a API e utilizá-lo quando fosse necessário.

Veja o diagrama UML abaixo:



(https://brizeno.files.wordpress.com/2011/10/adapter.png)

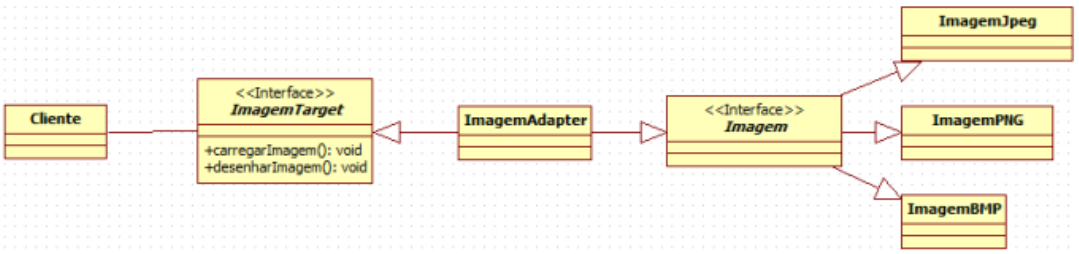
Um pouco de teoria

O exemplo acima foi bem irreal, pois implementamos as funcionalidades para várias APIs diferentes, o que necessitou um Adapter para cada. Num exemplo real seria necessário apenas um Adapter para a API utilizada, e apenas este Adapter sofreria mudanças quando necessário. A não ser que o projeto realmente precise utilizar várias APIs diferentes.

Os adaptadores podem ser definidos de duas maneiras. Na maneira mostrada no exemplo o adaptador adaptou a Classe, herdando tanto da interface comum quanto da classe da API. Outra maneira seria utilizar um adaptador de Objeto, nele o adaptador instancia um objeto da API e, nas chamadas de métodos repassa a chamada para este objeto.

Cada implementação de adapter possui suas implicações. Um adaptador de classe não vai conseguir trabalhar com as subclasses da classe adaptada, já ao utiliza um adaptador de objeto é possível trabalhar com objetos de subclasses da classe adaptada.

Por exemplo, considere que uma API de imagem possui uma classe para cada tipo de imagem (JPEG, PNG, BMP, etc..) e que estas classes possuem uma interface comum:



(https://brizeno.files.wordpress.com/2011/10/adapter-invic3a1vel3.png)

Utilizar um adaptador de classe não seria viável pois precisaríamos de um adaptador para cada subclasse, já que herdando de uma classe genérica não adicionaria nada.

Já utilizando um adaptador de objeto bastaria tem uma referência para um objeto da interface e instanciar qualquer uma das subclasses.

Quão mais próximas forem as responsabilidades das classes adaptadas, mais simples será o adaptador. No exemplo citado as duas classes (OpenGLImage e SDL_Surface) possuiam a mesma responsabilidade, assim cada adaptador precisou apenas realizar a chamada necessária. Daí surge um problema com o uso do adapter: a diferença de interface.

Caso a diferença de responsabilidades seja muito grande, um adaptador pode acabar por subutilizar uma determinada classe, limitando sua interface ao que é comum as outras classes adaptadas.

Outro aspecto foi que, utilizando o padrão Adapter, deixamos a API como uma espécie de "plug-in" do sistema. O nosso cliente trata apenas com a interface, ficando independente de qual API é utilizada. Esse foi justamente o ponto tratado no início deste post. Utilizando este padrão as regras de nogócio de um sistema ficam independentes de detalhes de implementação.

Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padr-es-de-Projeto (https://github.com/MarcosX/Padr-es-de-Projeto).

Os arquivos estão como um projeto do eclipse, então basta colocar no seu Workspace e fazer o import.

Se gostou do post compartilhe com seus amigos e colegas, senão, comente o que pode ser melhorado. Encontrou algum erro no código? Comente também. Possui alguma outra opinião ou alguma informação adicional? Comenta ai! 😀

Referências:

[1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

 Adapter, Padrões de Projeto Adapter, Java, Padrões, Projeto 4 Comentários

4 comentários sobre “Mão na massa: Adapter”

1. outubro 3, 2011 às 7:02 PM
   Yuri Adams
   É isso ai Marcos. Já está nos meus favoritos seu blog! Parabéns

    Responder
2. novembro 4, 2015 às 3:10 PM
   neycandidoribeiro
   Caramba, que super blog! Cheguei quatro anos atrasado, mas antes tarde do que nunca.

    Responder
3. abril 6, 2016 às 10:22 PM
   Anônimo
   Concordo com o colega acima, super blog mesmo!!! dicas valiosas e muito conhecimento existe aqui.

    Responder
4. maio 31, 2016 às 11:48 AM
   PAULO CESAR
   valeu

   também agradeço

    Responder

__PRESENT