Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

Facade

novembro 17, 2011

Mão na massa: Facade

Problema:

No desenvolvimento de jogos é comum a utilização de subsistemas de um Framework/API. Por exemplo, para reproduzir um determinado som é utilizado o subsistema de Audio, que normalmente provê funcionalidades desde a configuração da reprodução de audio, até a reprodução de um determinado arquivo.

Antes de iniciar o jogo de fato é necessário realizar ajustes em todos os subsistemas que serão utilizados, por exemplo, é necessário configurar a resolução do subsistema de Video para que este possa renderizar imagens corretamente.

Para exemplificar veja as interfaces da seguintes classes, que representa o subsistema de Audio:

```java
public class SistemaDeAudio {

    public void configurarFrequencia() {
        System.out.println("Frequencia configurada");
    }

    public void configurarVolume() {
        System.out.println("Volume configurado");
    }

    public void configurarCanais() {
        System.out.println("Canais configurados");
    }

    public void reproduzirAudio(String arquivo) {
        System.out.println("Reproduzindo: " + arquivo);
    }
}
```

Como falado ela fornece os métodos para configuração e reprodução de arquivos de audio. Para reproduzir um arquivo de audio, por exemplo, seria necessário realizar as seguintes operações:

```java
public static void main(String[] args) {
    System.out.println("##### Configurando subsistemas #####");
    SistemaDeAudio audio = new SistemaDeAudio();
    audio.configurarCanais();
    audio.configurarFrequencia();
    audio.configurarVolume();

    System.out.println("##### Utilizando subsistemas #####");
    audio.reproduzirAudio("teste.mp3");
}
```

Neste exemplo de código cliente, o próprio cliente deve instanciar e configurar o subsistema para que só depois seja possível a utilização dos mesmos. Além disso, existe um comportamento padrão que é executado antes de reproduzir um som: sempre deve ser configurado o canal, a frequência e o volume.

Agora pense como seria caso fosse necessário utilizar vários subsistemas? O código cliente ficaria muito sobrecarregado com responsabilidades que não são dele:

```java
public static void main(String[] args) {
    System.out.println("##### Configurando subsistemas #####");
    SistemaDeAudio audio = new SistemaDeAudio();
    audio.configurarCanais();
    audio.configurarFrequencia();
    audio.configurarVolume();

    SistemaDeInput input = new SistemaDeInput();
    input.configurarTeclado();
    input.configurarJoystick();

    SistemaDeVideo video = new SistemaDeVideo();
    video.configurarCores();
    video.configurarResolucao();

    System.out.println("##### Utilizando subsistemas #####");
    audio.reproduzirAudio("teste.mp3");
    input.lerInput();
    video.renderizarImagem("imagem.png");
}
```

Vamos ver então como o padrão Facade pode resolver este pequeno problema.

Facade

A intenção do padrão:

“Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Facade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.” [1]

Pela intenção é possível notar que o padrão pode ajudar bastante na resolução do nosso problema. O conjunto de interfaces seria exatamente o conjunto de subsistemas. Como falado em [1] um subsistema é análogo a uma classe, uma classe encapsula estados e operações, enquanto um subsistema encapsula classes.

Nesse sentido o Facade vai definir operações a serem realizadas com estes subsistemas. Assim, é possível definir uma operação padrão para configurar o subsistema de audio, evitando a necessidade de chamar os métodos de configuração de audio a cada novo arquivo de audio que precise ser reproduzido.

A utilização do padrão Facade é bem simples. apenas é necessário criar a classe fachada que irá se comunicar com os subsistemas no lugar no cliente:

```
1  public class SistemasFacade {
2      protected SistemaDeAudio audio;
3      protected SistemaDeInput input;
4      protected SistemaDeVideo video;
5
6      public void inicializarSubsistemas() {
7          video = new SistemaDeVideo();
8          video.configurarCores();
9          video.configurarResolucao();
10
11         input = new SistemaDeInput();
12         input.configurarJoystick();
13         input.configurarTeclado();
14
15         audio = new SistemaDeAudio();
16         audio.configurarCanais();
17         audio.configurarFrequencia();
18         audio.configurarVolume();
19     }
20
21     public void reproduzirAudio(String arquivo) {
22         audio.reproduzirAudio(arquivo);
23     }
24
25     public void renderizarImagem(String imagem) {
26         video.renderizarImagem(imagem);
27     }
28
29     public void lerInput() {
30         input.lerInput();
31     }
32
33 }
```

A classe fachada realiza a inicialização de todos os subsistemas e oferece acesso aos métodos necessários, por exemplo o método de renderização de uma imagem, a reprodução de um audio.

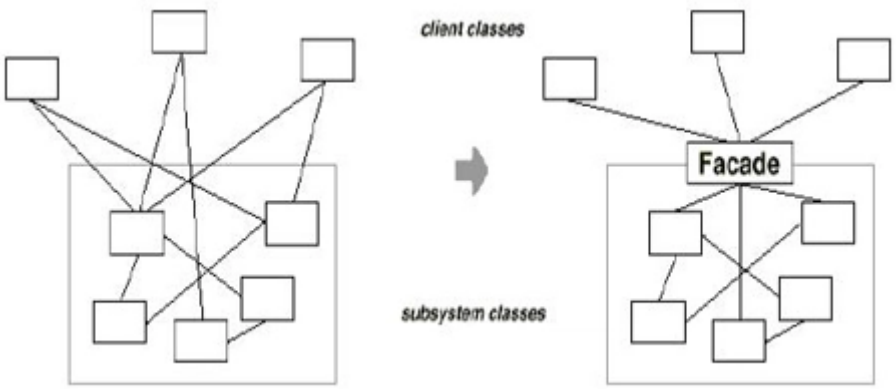
Com esta mudança, tiramos toda a responsabilidade do cliente, que agora precisa se preocupar apenas em utilizar os subsistemas que desejar.

```
1  public static void main(String[] args) {
2      System.out.println("##### Configurando subsistemas #####");
3      SistemasFacade fachada = new SistemasFacade();
4      fachada.inicializarSubsistemas();
5
6      System.out.println("##### Utilizando subsistemas #####");
7      fachada.renderizarImagem("imagem.png");
8      fachada.reproduzirAudio("teste.mp3");
9      fachada.lerInput();
10 }
```

A utilização do padrão é bem simples e poder ser aplicado em várias situações.

Um pouco de teoria

A ideia básica do padrão, como visto, é remover a complexidade das classes clientes. Visualmente falando, ele faz o seguinte:



Note que as classes do subsistema continuam sendo visíveis em todo o projeto. Portanto, caso seja necessário, o cliente pode definir suas configurações sem sequer utilizar a classe fachada. Ainda mais, o cliente pode criar uma fachada própria, que define suas operações customizadas. Por exemplo, se não serão utilizados joysticks no projeto, não há necessidade de inicializá-los.

O problema com essa centralização da complexidade é que a classe fachada pode crescer descontroladamente para abrigar uma conjunto grande de possibilidades. Nestes casos pode ser mais viável procurar outros padrões, como Abstract Factory para dividir as responsabilidades entre subclasses.

Existem algumas semelhanças, fáceis de serem notadas, deste padrão com outros já discutidos aqui. Por exemplo, já que o Facade define uma interface, qual a sua diferença em relação ao padrão Adapter, já que ambos definem uma nova interface? A diferença básica é que o Adapter adapta uma interface antiga para uma outra interface enquanto que o Facade cria uma interface completamente nova, que é mais simples.

Outra semelhança que também pode ser notada é com o padrão Mediator, já que ambos reduzem a complexidade entre um grande conjunto de objetos. A diferença é que como o padrão Mediator centraliza a comunicação entre os objetos colegas, normalmente adicionando novas funcionalidades e sendo referenciado por todos os colegas. No padrão Facade, a classe fachada apenas utiliza os subsistemas, sem adicionar nada, além disso as classes do subsistema não sabem nada sobre a fachada.

Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: 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.

 Facade Facade, Java, Padrões, Projeto 5 Comentários

__PRESENT