Marcos Brizeno

Desenvolvimento de Software #showmethecode

Flyweight

novembro 13, 2011

Mão na massa: Flyweight

Problema:

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.

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 entende 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 de 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:

```
1  public class Imagem {
2      protected String nomeDaImagem;
3
4      public Imagem(String imagem) {
5          nomeDaImagem = imagem;
6      }
7
8      public void desenharImagem() {
9          System.out.println(nomeDaImagem + " desenhada!");
10     }
11 }
```

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

```
1  public class Ponto {
2      public int x, y;
3
4      public Ponto(int x, int y) {
5          this.x = x;
6          this.y = y;
7      }
8  }
```

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

```
1  public abstract class SpriteFlyweight {
2      public abstract void desenharImagem(Ponto ponto);
3  }
```

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();
```

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.

```
public class FlyweightFactory {

    protected ArrayList<SpriteFlyweight> flyweights;

    public enum Sprites {
        JOGADOR, INIMIGO_1, INIMIGO_2, INIMIGO_3, CENARIO_1, CENARIO_
    }

    public FlyweightFactory() {
        flyweights = new ArrayList<SpriteFlyweight>();
        flyweights.add(new Sprite("jogador.png"));
        flyweights.add(new Sprite("inimigo1.png"));
        flyweights.add(new Sprite("inimigo2.png"));
        flyweights.add(new Sprite("inimigo3.png"));
        flyweights.add(new Sprite("cenario1.png"));
        flyweights.add(new Sprite("cenario2.png"));
    }

    public SpriteFlyweight getFlyweight(Sprites jogador) {
        switch (jogador) {
        case JOGADOR:
            return flyweights.get(0);
        case INIMIGO_1:
            return flyweights.get(1);
        case INIMIGO_2:
            return flyweights.get(2);
        case INIMIGO_3:
            return flyweights.get(3);
        case CENARIO_1:
            return flyweights.get(4);
        default:
            return flyweights.get(5);
        }
    }
}
```

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.

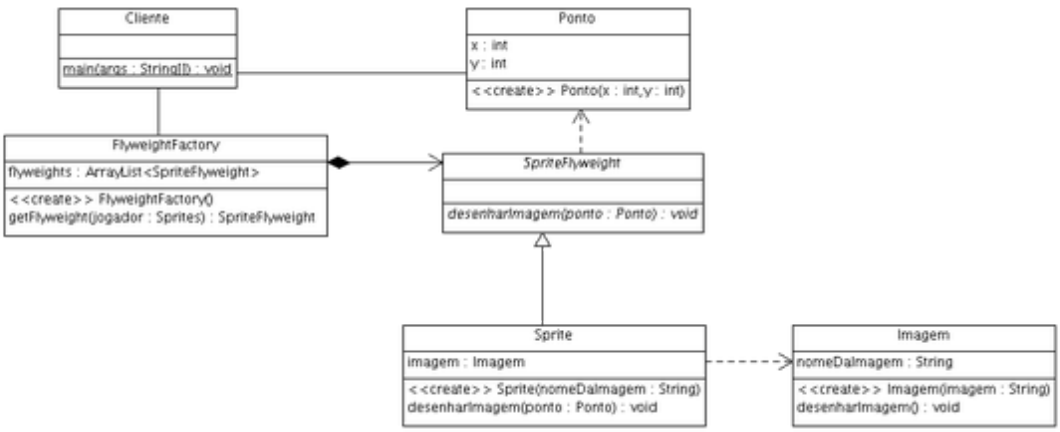
Para exemplificar a utilização do padrão, vejamos o seguinte código cliente:

```
1 | public static void main(String[] args) {
```

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.

O diagrama UML que representa esta implementação é o seguinte:



Um pouco de teoria

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.

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.

Flyweight Flyweight, Java, Padrões, Projeto Anúncios 2 Comentários