Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

State

novembro 21, 2011

Mão na massa: State

Problema:

A troca de estados de um objeto é um problema bastante comum. Tome como exemplo o personagem de um jogo, como o Mario. Durante o jogo acontecem várias trocas de estado com o Mario, por exemplo, ao pegar uma flor de fogo o mario pode crescer, se estiver pequeno, e ficar com a habilidade de soltar bolas de fogo.

Desenvolvendo um pouco mais o pensamento temos um conjunto grande de possíveis estados, e cada transição depende de qual é o estado atual do personagem. Como falado anteriormente, ao pegar uma flor de fogo podem acontecer quatro ações diferentes, dependendo de qual o estado atual do mario:

Se Mario pequeno -> Mario grande e Mario fogo
Se Mario grande -> Mario fogo
Se Mario fogo -> Mario ganha 1000 pontos
Se Mario capa -> Mario fogo

Todas estas condições devem ser checadas para realizar esta única troca de estado. Agora imagine o vários estados e a complexidade para realizar a troca destes estados: Mario pequeno, Mario grande, Mario flor e Mario pena.

Pegar Cogumelo:
Se Mario pequeno -> Mario grande
Se Mario grande -> 1000 pontos

Se Mario fogo -> 1000 pontos
Se Mario capa -> 1000 pontos

Pegar Flor:
Se Mario pequeno -> Mario grande e Mario fogo
Se Mario grande -> Mario fogo
Se Mario fogo -> 1000 pontos
Se Mario capa -> Mario fogo

Pegar Pena:
Se Mario pequeno -> Mario grande e Mario capa
Se Mario grande -> Mario capa
Se Mario fogo -> Mario fogo
Se Mario capa -> 1000 pontos

Levar Dano:
Se Mario pequeno -> Mario morto
Se Mario grande -> Mario pequeno
Se Mario fogo -> Mario grande
Se Mario capa -> Mario grande

Com certeza não vale a pena investir tempo e código numa solução que utilize várias verificações para cada troca de estado. Para não correr o risco de esquecer de tratar algum estado e deixar o código bem mais fácil de manter, vamos analisar como o padrão State pode ajudar.

State

A intenção do padrão:

“Permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.” [1]

Pela intenção podemos ver que o padrão vai alterar o comportamento de um objeto quando houver alguma mudança no seu estado interno, como se ele tivesse mudado de classe.

Para implementar o padrão será necessário criar uma classe que contém a interface básica de todos os estados. Como definimos anteriormente o que pode causar alteração nos estados do objeto Mario, estas serão as operações básicas que vão fazer parte da interface.

```
1  public interface MarioState {
2      MarioState pegarCogumelo();
3
4      MarioState pegarFlor();
5
6      MarioState pegarPena();
7
8      MarioState levarDano();
9  }
```

Agora todos os estados do mario deverão implementar as operações de troca de estado. Note que cada operação retorna um objeto do tipo MarioState, pois como cada operação representa uma troca de estados, será retornado qual o novo estado o mario deve assumir.

Vejamos então uma classe para exemplificar um estado:

```
1  public class MarioPequeno implements MarioState {
2
3      @Override
4      public MarioState pegarCogumelo() {
5          System.out.println("Mario grande");
6          return new MarioGrande();
7      }
8
9      @Override
10     public MarioState pegarFlor() {
11         System.out.println("Mario grande com fogo");
12         return new MarioFogo();
13     }
14
15     @Override
16     public MarioState pegarPena() {
17         System.out.println("Mario grande com capa");
18         return new MarioCapa();
19     }
20
21     @Override
22     public MarioState levarDano() {
23         System.out.println("Mario morto");
24         return new MarioMorto();
25     }
26
27 }
```

Percebemos que a classe que define o estado é bem simples, apenas precisa definir qual estado deve ser trocado quando uma operação de troca for chamada. Vejamos agora outro exemplo de classe de estado:

```java
public class MarioCapa implements MarioState {

    @Override
    public MarioState pegarCogumelo() {
        System.out.println("Mario ganhou 1000 pontos");
        return this;
    }

    @Override
    public MarioState pegarFlor() {
        System.out.println("Mario com fogo");
        return new MarioFogo();
    }

    @Override
    public MarioState pegarPena() {
        System.out.println("Mario ganhou 1000 pontos");
        return this;
    }

    @Override
    public MarioState levarDano() {
        System.out.println("Mario grande");
        return new MarioGrande();
    }

}
```

Bem simples não? Novos estados são adicionados de maneira bem simples. Vejamos então como seria o objeto que vai utilizar o estados, o Mario:

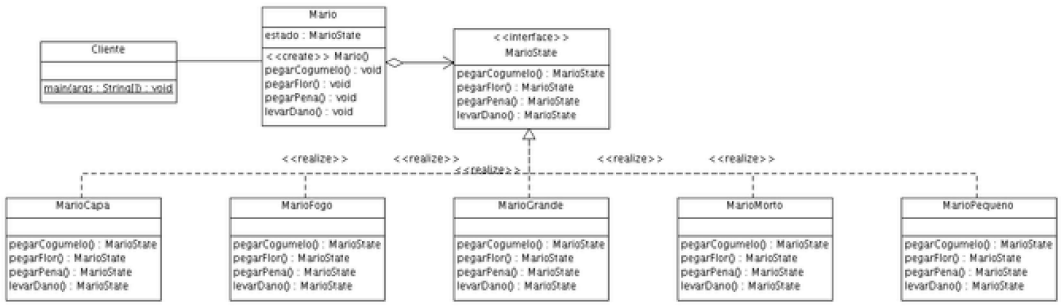
```java
public class Mario {
    protected MarioState estado;

    public Mario() {
        estado = new MarioPequeno();
    }

    public void pegarCogumelo() {
        estado = estado.pegarCogumelo();
    }

    public void pegarFlor() {
        estado = estado.pegarFlor();
    }

    public void pegarPena() {
        estado = estado.pegarPena();
    }

    public void levarDano() {
        estado = estado.levarDano();
    }
}
```

A classe mario possui uma referência para um objeto estado, este estado vai ser atualizado de acordo com as operações de troca de estados, definidas logo em seguida. Quando uma operação for invocada, o objeto estado vai executar a operação e se atualizará automaticamente. Como exemplo de utilização, vejamos o seguinte código cliente:

```
1  public static void main(String[] args) {
2      Mario mario = new Mario();
3      mario.pegarCogumelo();
4      mario.pegarPena();
5      mario.levarDano();
6      mario.pegarFlor();
7      mario.pegarFlor();
8      mario.levarDano();
9      mario.levarDano();
10     mario.pegarPena();
11     mario.levarDano();
12     mario.levarDano();
13     mario.levarDano();
14 }
```

Por este código é possível avaliar todas as transições e todos os estados.

O diagrama UML a seguir resume visualmente as relações entre as classes:



Um pouco de teoria

Pelo visto no exemplo, o padrão é utilizado quando se precisa isolar o comportamento de um objeto, que depende de seu estado interno. O padrão elimina a necessidade de condicionais complexos e que frequentemente serão repetidos. Com o padrão cada "ramo" do condicional acaba se tornando um objeto, assim você pode tratar cada estado como se fosse um objeto de verdade, distribuindo a complexidade dos condicionais.

Incluir novos estados também é muito simples, basta criar uma nova classe e atualizar as operações de transição de estados. Com a primeira solução seriam necessários vários milhões de ifs novos e a alteração dos já existentes, além do grande risco de esquecer algum estado. Outra grande vantagem é que fica claro, com a estrutura do padrão, quais são os estados e quais são as possíveis transições.

O padrão State não define aonde as transições ocorrem, elas podem ser colocadas dentro das classes de estado ou dentro da classe que armazena o estado. No exemplo vimos que dentro de cada estado são definidos os novos objetos que são retornados. A principal vantagem desta solução é que fica mais simples adicionar os estados, cada novo estado define suas transições. O problema é que assim cada classe de estado precisa ter conhecimento sobre as outras subclasses, e se alguma delas mudar, é provável que a mudança se espalhe.

É comum que objetos estados obedeçam a outros dois padrões: Singleton e Flyweight. Estados singleton são capazes de manter informações, mesmo com as constantes trocas de estados. Estados Flyweight permitem o compartilhamento entre objetos que vão utilizar a mesma máquina de estado.

O padrão State tem semelhanças com outros dois padrões: Strategy e Bridge. Falando primeiro sobre o Strategy, note que a ideia é muito parecida: eliminar vários ifs complexos espalhados utilizando subclasses. A diferença básica é que o State é mais dinâmico que o Strategy, pois ocorrem várias trocas de objetos estados, os próprios objetos estados realizam as transições.

A semelhança com o padrão Bridge também pode ser notada facilmente pelo diagrama UML, no entanto a diferença está na intenção dos padrões. A intenção do Bridge é permitir que tanto a implementação quanto a interface possam mudar independentemente. No State a ideia é realmente mudar o comportamento de um objeto.

Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padroes-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.

State Java, Padrões, Projeto, State 10 Comentários

__PRESENT