



INSTITUTO FEDERAL

Rio Grande do Sul
Campus Rio Grande



State

Prof. Igor Avila Pereira
igoravilapereira.github.io
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande
Divisão de Computação

Agenda

- 1 Introdução
- 2 Problema
- 3 State
- 4 Prós e Contras
- 5 Um pouco de teoria
- 6 Trabalho

Introdução

Um fato pouco conhecido: os Padrões State e Strategy são irmãos gêmeos que foram separados ao nascer.

- Como você sabe, o Padrão Strategy acabou se dando muito bem no negócio dos algoritmos intercambiáveis.
- O Padrão State, por sua vez, torna é mais nobre no que tange ajudar objetos a controlarem seu comportamento através de mudanças no seu estado interno

Problema

- A troca de estados de um objeto é um problema bastante comum. Tome como exemplo o personagem de um jogo, como o Mário.



Problema

- Durante o jogo acontecem várias trocas de estado com o Mário, por exemplo, ao pegar uma flor de fogo o Mário 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.

Problema

Como falado anteriormente, ao pegar uma flor de fogo podem acontecer quatro ações diferentes, dependendo de qual o estado atual do Mário:

- Se Mário pequeno \implies Mário grande e Mário fogo
- Se Mário grande \implies Mário fogo
- Se Mário fogo \implies Mário ganha 1000 pontos
- Se Mário capa \implies Mário fogo

Problema

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:

- Mário pequeno
- Mário grande
- Mário flor e
- Mário pena.

Problema

Pegar Cogumelo:

- Se Mário pequeno \implies Mário grande
- Se Mário grande \implies 1000 pontos
- Se Mário fogo \implies 1000 pontos
- Se Mário capa \implies 1000 pontos



Problema

Pegar Flor:

- Se Mário pequeno \implies Mário grande e Mário fogo
- Se Mário grande \implies Mário fogo
- Se Mário fogo \implies 1000 pontos
- Se Mário capa \implies Mário fogo



Problema

Pegar Pena:

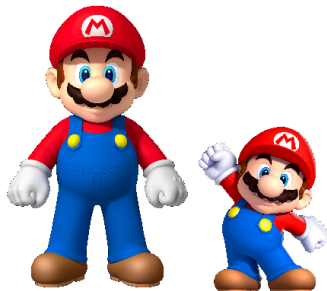
- Se Mário pequeno \implies Mário grande e Mário capa
- Se Mário grande \implies Mário capa
- Se Mário fogo \implies Mário fogo
- Se Mário capa \implies 1000 pontos



Problema

Levar Dano:

- Se Mário pequeno \implies Mário morto
- Se Mário grande \implies Mário pequeno
- Se Mário fogo \implies Mário grande
- Se Mário capa \implies Mário grande



Problema

- Com certeza não vale a pena investir tempo e código numa solução que utilize várias verificações (**ifs**) 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

- 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.

State

- 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 Mário, estas serão as operações básicas que vão fazer parte da interface.

State



Figura: Estados Mário

State

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


State

- Agora todos os estados do Mário 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 Mário deve assumir.

State

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 }
```

State

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

State

Vejamos agora outro exemplo de classe de estado:

```
1 public class MarioCapa implements MarioState {  
2  
3     @Override  
4     public MarioState pegarCogumelo() {  
5         System.out.println("Mario ganhou 1000 pontos");  
6         return this;  
7     }  
8  
9     @Override  
10    public MarioState pegarFlor() {  
11        System.out.println("Mario com fogo");  
12        return new MarioFogo();  
13    }  
14  
15    @Override  
16    public MarioState pegarPena() {  
17        System.out.println("Mario ganhou 1000 pontos");  
18        return this;  
19    }  
20  
21    @Override  
22    public MarioState levarDano() {  
23        System.out.println("Mario grande");  
24        return new MarioGrande();  
25    }  
26 }  
27 }
```

State

Bem simples não?

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

```
1 public class Mario {  
2     protected MarioState estado;  
3  
4     public Mario() {  
5         estado = new MarioPequeno();  
6     }  
7  
8     public void pegarCogumelo() {  
9         estado = estado.pegarCogumelo();  
10    }  
11  
12    public void pegarFlor() {  
13        estado = estado.pegarFlor();  
14    }  
15  
16    public void pegarPena() {  
17        estado = estado.pegarPena();  
18    }  
19  
20    public void levarDano() {  
21        estado = estado.levarDano();  
22    }  
23 }
```

Prós e Contras

- 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.

State

- A classe Mário 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.

State

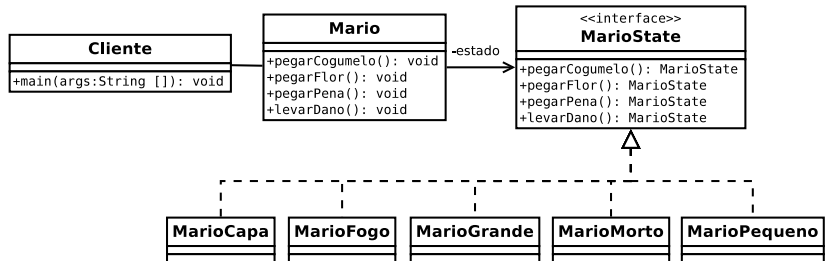
Como exemplo de utilização, vejamos o seguinte código cliente:

```
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 }
```

Este código permite avaliar todas as transições e todos os estados

State

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



Um pouco de teoria....

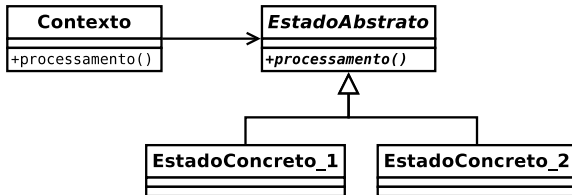


Figura: UML genérico do padrão

Um pouco de teoria....

- **Contexto:** é a classe que pode ter vários estados internos diferentes.
- **Estado:** define uma interface comum para todos os estados concretos. Como são intercambiáveis, todos devem implementar a mesma interface
- **Estados Concretos:** lidam com as solicitações provenientes do Contexto.
 - Cada Estado concreto fornece a sua própria implementação de uma solicitação Assim, quando o Contexto muda de estado, seu comportamento também muda.

Um pouco de teoria...

- **Espera aí:** se estou me lembrando bem o Padrão Strategy, este diagrama de classe é EXATAMENTE o mesmo.
- Você tem um bom olho! Sim, os diagramas de classes são essencialmente os mesmos, mas os dois padrões diferem nas suas **intenções**.

Um pouco de teoria...

State

Com o padrão State, temos um conjunto de comportamentos encapsulados em objetos de estado:

- a qualquer momento, o contexto está delegando tarefas a um desses estados.
- Com o tempo, o estado atual muda para um dos objetos de estado para refletir o estado interno do contexto;
 - **Portanto:** o comportamento do próprio contexto também muda ao longo do tempo.
 - O cliente geralmente sabe muito pouco ou nada sobre os objetos de estado.

Um pouco de teoria....

Strategy

Com Strategy, o cliente geralmente especifica o objeto de estratégia com o qual o contexto é composto.

- Neste caso, embora o padrão proporcione a flexibilidade necessária para o objeto de estratégia durante a execução, frequentemente há um objeto de estratégia que é o mais indicado para o objeto de contexto

Um pouco de teoria...

Via de regra:

Deve-se pensar no Padrão Strategy como uma alternativa flexível à criação de subclasses:

- quando você usa a hereditariedade para definir o comportamento de uma classe, o resultado é um comportamento imutável mesmo que seja necessário introduzir alterações.
- **Você pode mudar o comportamento estabelecendo uma composição com um objeto diferente**

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.

Um pouco de teoria...

- 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.

Um pouco de teoria...

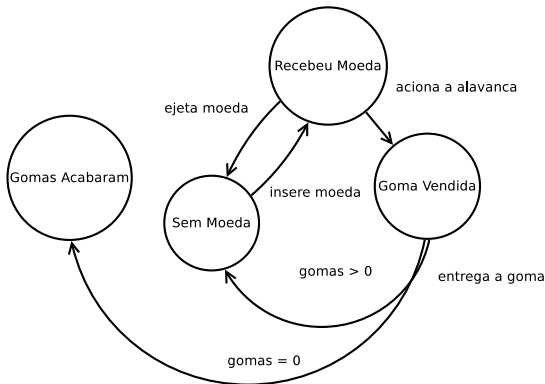
- O padrão State não define aonde as transições ocorrem, elas podem ser colocadas dentro das classes de estado ou dentro da classe que armazena o estado.
- No exemplo vimos que dentro de cada estado são definidos os novos objetos que são retornados.

Trabalho

- Hoje em dia as pessoas estão usando Java para construir dispositivos reais, como as máquinas de goma de mascar.
- É isso mesmo, as máquinas que vem goma de mascas aderiram à alta tecnologia.
- Os principais fabricantes descobriram que a inclusão de uma CPU nesses dispositivos aumenta as vendas, permite monitorar o estoque através da rede e ajuda a medir com mais precisão a satisfação dos clientes.

Trabalho

Aqui está como nós achamos que o controlador da máquina de goma de mascar deveria funcionar.



Trabalho

Descrição

Esperamos que você consiga implementar isto em JAVA para nós! Pretendemos acrescentar outros comportamentos no futuro; portanto, você deve manter o projeto tão flexível e fácil de manter quanto seja possível!

Implemente o padrão State para o problema proposto

Trabalho

Dica:

Como podemos transplantar o diagrama de estados para código de programação?

- 1 Em primeiro lugar, determine quais são os estados
- 2 Em seguida, crie uma variável de estado
- 3 Determine as ações que podem ocorrer no sistema: **insere moeda, ejeta a moeda, aciona a alavanca e entrega a goma**
- 4 Defina uma classe que atue com máquina de estados



INSTITUTO FEDERAL

Rio Grande do Sul
Campus Rio Grande



State

Prof. Igor Avila Pereira
igoravilapereira.github.io
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande
Divisão de Computação