State

Padrões Comportamentais

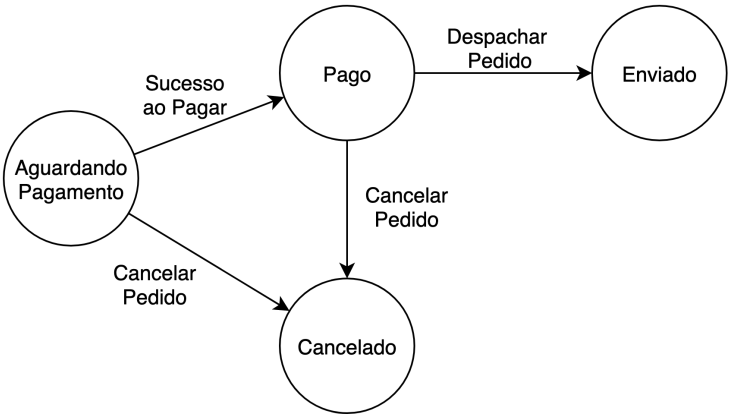
O padrão de projeto *State* permite que um objeto altere o seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe.

Motivação (Por que utilizar?)

Em alguns contextos de desenvolvimento de *software* existem objeto que se comportam de forma diferente conforme o estado em que se encontram. Nestes casos pode ser muito trabalhoso gerenciar tais mudanças de estado, além de gerar classes enormes, com diversas operações condicionais que definem se é possível migrar de um estado para outro.

É comum encontrar classe com um atributo **estado** que utilizam valores inteiros ou constantes para representar o estado interno em que um objeto se encontra. Toda a lógica de transição entre estados costuma ficar na própria classe:

Considere um pedido em um *e-commerce* onde tal pedido pode passar pelos seguintes estados:



Máquina de estados

A máquina de estados nos diz que:

- Um pedido é inicializado no estado Aguardando Pagamento.
- Um pedido pode ir para o estado Pago, se e somente se, estiver no estado Aguardando pagamento e a ação Sucesso ao Pagar for solicitada.
- Um pedido pode ir para o estado Cancelado, se e somente se, estiver nos estados Aguardando pagamento ou Pago e a ação Cancelar Pedido for solicitada.
- Um pedido pode ir para o estado Enviado, se e somente se estiver no estado Pago e a ação Despachar Pedido for solicitada.

Uma possível implementação sem o padrão *State* seria assim:

```
class Pedido
{
    //Constantes que definem os possíveis estados do pedido
    const AGUARDANDO_PAGAMENTO = 1;
    const PAGO = 2;
    const CANCELADO = 3;
    const ENVIADO = 4;

    private int $estadoAtual; //Estado atual do pedido.

    public function __construct()
    {
        //Definição do estado inicial do pedido.
        $this->estadoAtual = self::AGUARDANDO_PAGAMENTO;
    }

    //Transição Sucesso ao Pagar;
    public function sucessoAoPagar()
    {
        //Se o pedido estiver aguardando pagamento.
        if ($this->estadoAtual == self::AGUARDANDO_PAGAMENTO) {
            //Mude o estado do pedido para Pago.
            $this->estadoAtual = self::PAGO;
        } else {
            //Senão mostre uma mensagem de erro;
            echo 'O pedido não está aguardando pagamento';
        }
    }

    //Transições Cancelar Pedido;
    public function cancelarPedido()
    {
        //Se o pedido estiver aguardando pagamento;
        if ($this->estadoAtual == self::AGUARDANDO_PAGAMENTO) {
            //Mude o estado do pedido para Cancelado;
            $this->estadoAtual = self::CANCELADO;
        //Se o pedido estiver Pago;
        } elseif ($this->estadoAtual == self::PAGO) {
            //Mude o estado do pedido para Cancelado;
            $this->estadoAtual = self::CANCELADO;
        } else {
            echo 'O pedido não pode ser cancelado'; //Senão mostre uma mensagem de erro;
        }
    }

    //Transição Despachar Pedido;
    public function despacharPedido()
    {
        //Se o pedido estiver Pago.
        if ($this->estadoAtual == self::PAGO) {
            //Mude o estado do pedido para Enviado.
            $this->estadoAtual = self::ENVIADO;
        } else {
            echo 'O pedido se encontra cancelado'; //Senão mostre uma mensagem de erro;
        }
    }
}
```

O padrão *state* sugere que cada um desses estados se torne um objeto de estado que irá compor o objeto de contexto (instância da classe **Pedido**). Um objeto de contexto é aquele que muda seu estado conforme o contexto em que se encontra, e por consequência muda seu comportamento. Deste modo o objeto de contexto parece ser uma instância de outra classe (A classe **Pedido** parecerá ter mudado de código), porém, tal mudança acontece devido a alteração do objeto de estado que o compõem.

Todos os objetos de estado devem assinar um contrato em comum, seja ele uma classe abstrata ou interface que contenha as solicitações que causam as mudanças de estado do objeto de contexto (**Pedido**). Isso garante a classe **Pedido** que todos os objetos de estado terão implementado os métodos (solicitações) que causam mudança no estado interno do pedido.

Cada transição se torna um método da interface **State**.

```
interface State
{
    //Transição Sucesso ao Pagar;
    public function sucessoAoPagar(): void;

    //Transições Cancelar Pedido;
    public function cancelarPedido(): void;

    //Transição Despachar Pedido;
    public function despacharPedido(): void;
}
```

Cada estado do pedido se torna uma classe que implementa a interface **State**.

```
class AguardandoPagamentoState implements State
{
    //Refência a classe Pedido.
    private Pedido $pedido;

    public function __construct(Pedido $pedido)
    {
        //Guarda referência do pedido.
        $this->pedido = $pedido;
    }

    //Transição Sucesso ao Pagar;
    public function sucessoAoPagar(): void
    {
        /*Ao ter sucesso ao pagar um Pedido Aguardando Pagamento mudar o estado do
          pedido para Pago.*/
        $this->pedido->setEstadoAtual($this->pedido->getPago());
    }
```

```
    //Transições Cancelar Pedido;
    public function cancelarPedido(): void
    {
        //Ao tentar cancelar um pedido Aguardando Pagamento, mostrar mensagem de erro.
        throw new \Exception('Operação não suportada, o pedido ainda não foi pago.');
    }

    //Transição Despachar Pedido;
    public function despacharPedido(): void
    {
        //Ao tentar despachar um pedido Aguardando Pagamento, mostrar mensagem de erro.
        throw new \Exception('Operação não suportada, o pedido ainda não foi pago.');
    }
}
```

As demais classes de estado seguem a mesma lógica que a classe `AguardandoPagamentoState`.

```
class PagoState implements State
{
    private Pedido $pedido;

    public function __construct(Pedido $pedido)
    {
        $this->pedido = $pedido;
    }

    public function sucessoAoPagar(): void
    {
        throw new \Exception('Operação não suportada, o pedido já foi pago.');
    }

    public function cancelarPedido(): void
    {
        $this->pedido->setEstadoAtual($this->pedido->getCancelado());
    }


    public function despacharPedido(): void
    {
        $this->pedido->setEstadoAtual($this->pedido->getEnviado());
    }
}
```

```php
class CanceladoState implements State
{
    private Pedido $pedido;

    public function __construct(Pedido $pedido)
    {
        $this->pedido = $pedido;
    }

    public function sucessoAoPagar(): void
    {
        throw new \Exception('Operação não suportada, o pedido se encontra cancelado.');
    }

    public function cancelarPedido(): void
    {
        throw new \Exception('Operação não suportada, pedido já cancelado.');
    }

    public function despacharPedido(): void
    {
        throw new \Exception('Operação não suportada, o pedido se encontra cancelado.');
    }
}
```

```php
class EnviadoState implements State
{
    private Pedido $pedido;

    public function __construct(Pedido $pedido)
    {
        $this->pedido = $pedido;
    }

    public function sucessoAoPagar(): void
    {
        throw new \Exception('Operação não suportada, o pedido já foi pago e enviado.');
    }

    public function cancelarPedido(): void
    {
        throw new \Exception('Operação não suportada, o pedido já foi enviado.');
    }

    public function despacharPedido(): void
    {
        throw new \Exception('Operação não suportada, o pedido já foi enviado.');
    }
}
```

Dada a existência das classes de estado, quando uma determinada solicitação tenta mudar o atual estado interno do objeto de contexto (**Pedido**), ele delega para seu atual objeto de estado a responsabilidade de como tal solicitação deve ser tratada.

Vejamos como fica a classe Pedido com os estados em objetos separados.

```
class Pedido
{
    //mantém uma referência para cada objeto de estado.
    private State $aguardandoPagamento;
    private State $pago;
    private State $cancelado;
    private State $enviado;

    //Estado atual do pedido.
    private State $estadoAtual;

    public function __construct()
    {
        //Criação e atribuição dos possíveis estados de Pedido.
        $this->aguardandoPagamento = new AguardandoPagamentoState($this);
        $this->pago = new PagoState($this);
        $this->cancelado = new CanceladoState($this);
        $this->enviado = new EnviadoState($this);
        //Definição do estado atual.
        $this->estadoAtual = $this->aguardandoPagamento;
    }

    //Transição Sucesso ao Pagar;
    public function sucessoAoPagar()
    {
        try {
            //Chama o método sucessoAoPagar() do objeto de estado atual.
            $this->estadoAtual->sucessoAoPagar();
        } catch (\Exception $e) {
            echo $e->getMessage();
        }
    }

    //Transições Cancelar Pedido;
    public function cancelarPedido()
    {
        try {
            //Chama o método cancelarPedido() do objeto de estado atual.
            $this->estadoAtual->cancelarPedido();
        } catch (\Exception $e) {
            echo $e->getMessage();
        }
    }
```

```
    //Transição Despachar Pedido;
    public function despacharPedido()
    {
        try {
            //Chama o método despacharPedido() do objeto de estado atual.
            $this->estadoAtual->despacharPedido();
        } catch (\Exception $e) {
            echo $e->getMessage();
        }
    }

    /*Getters que permitem que as classes de estado recuperarem os possíveis
      objetos de estado de Pedido*/

    public function getAguardandoPagamento(): State
    {
        return $this->aguardandoPagamento;
    }

    public function getPago(): State
    {
        return $this->pago;
    }

    public function getCancelado(): State
    {
        return $this->cancelado;
    }

    public function getEnviado(): State
    {
        return $this->enviado;
    }

    //Setter que permite que os objetos de estados troquem o estado atual de Pedido.
    public function setEstadoAtual(State $estadoAtual): void
    {
        $this->estadoAtual = $estadoAtual;
    }
}
```

As classes de estado facilitam o gerenciamento das transições entre os estados de **Pedido**, previne que a classe cresça demais, e se torne difícil de entender e ainda simplifica o processo de inserção de novos estados.

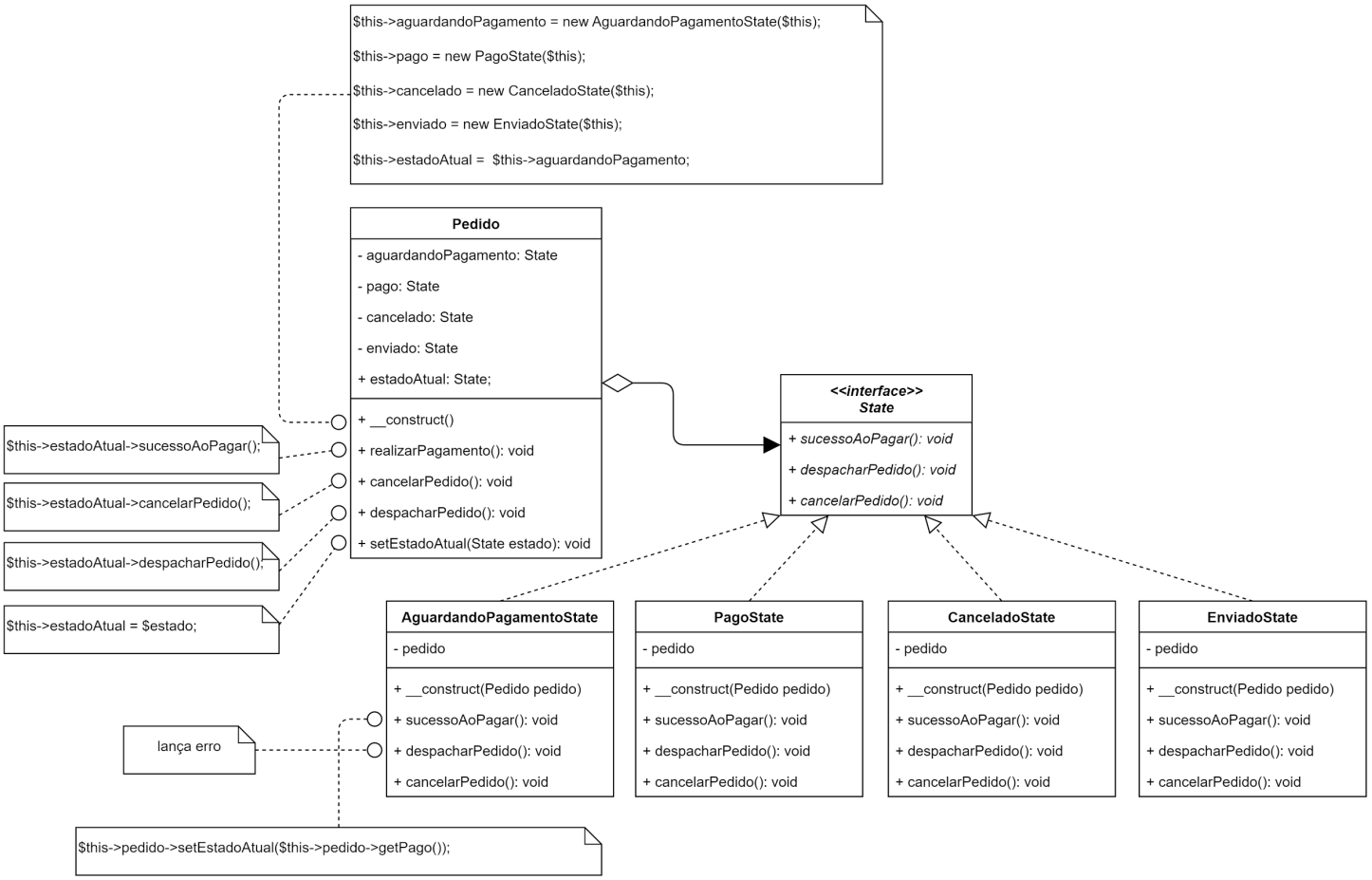


Diagrama de classes do exemplo (Getters foram ocultados)

Aplicabilidade (Quando utilizar?)

- Quando o comportamento de um objeto depende do seu estado interno, e com base nele muda seu comportamento em tempo de execução.

- Quando operações possuírem instruções condicionais grandes que dependam do estado interno do objeto. Frequentemente várias destas operações terão a mesmas estruturas condicionais.

Componentes

- **Contexto:** É a classe que pode ter vários estados internos diferentes. Ela mantém uma instância de uma subclasse EstadoConcreto que define seu estado interno atual. Sempre que uma solicitação é feita ao contexto, ela é delegada ao estado atual para ser processada.

- **State:** Define uma interface (ou classe abstrata) comum para todos os estados concretos.

- **EstadoConcreto:** Lidam com as solicitações provenientes do contexto. Cada EstadoConcreto fornece a sua própria implementação de uma solicitação. Deste modo, quando o contexto muda de estado interno o seu comportamento também muda.

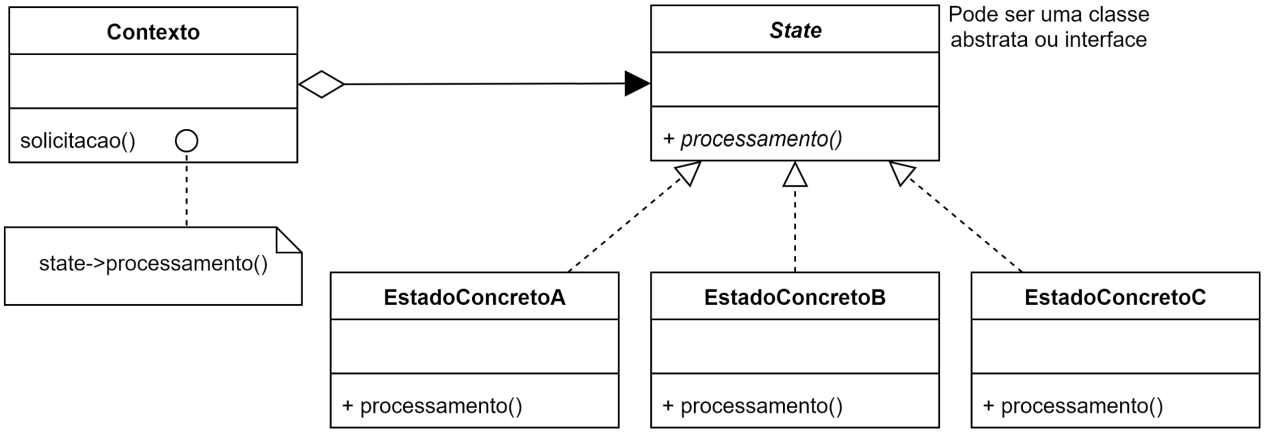


Diagrama de Classes

Consequências

- O padrão *State* encapsula o comportamento específico de um estado, e como o objeto de contexto deve se comportar em cada estado. O padrão coloca todo o comportamento associado a um estado específico em um objeto separado, assim, todo código referente a tal comportamento fica em uma subclasse EstadoConcreto. Novos estados podem ser adicionados facilmente, apenas definindo novas subclasses EstadoConcreto.

- As transições de estado se tornam explícitas. Quando um objeto define seu estado atual apenas em termos de valores de dados internos (constantes ou inteiros), suas transições de estado não têm representação explícita. Tais valores aparecem somente como atribuições para algumas variáveis.

- Estados podem proteger seu contexto interno de transições de inconsistências, porque as transições são processadas a nível de estado e não no objeto de contexto.