State

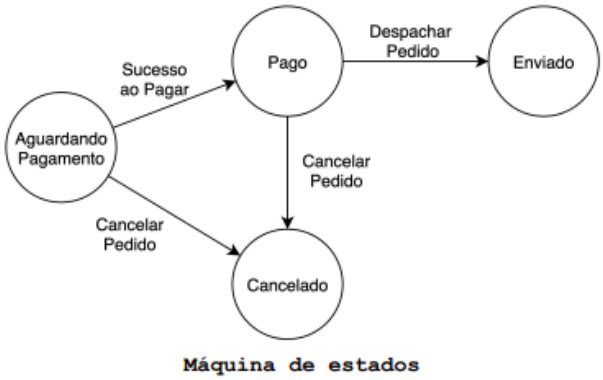
Definição

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

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:

```
export class Order {

  private awaitingPayment = 1
  private payment = 2
  private canceled = 3
  private sent = 4

  private state: number

  constructor() {
    this.state = this.awaitingPayment
  }

  successWhenPaying() {
    if (this.state !== this.awaitingPayment) {
      throw new Error('The order is awaiting payment');
    }
    this.state = this.payment
  }

  cancelOrder() {
    if (this.state === this.awaitingPayment) {
      this.state = this.canceled
      return
    }
    if (this.state === this.payment) {
      this.state = this.canceled
      return
    }
    throw new Error('The order cannot be canceled')
  }

  dispatchOrder() {
    if(this.state === this.payment) {
      this.state = this.sent
      return
    }

    throw new Error('The order is canceled')
  }

}
```

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`.

```
export interface State {
  successWhenPaying(): void
  cancelOrder(): void
  dispatchOrder(): void
}
```

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

```
import { State } from '../interface/state'
import { Order } from './order'

export class AwaitingPayment implements State {


  constructor(private order: Order){}

  successWhenPaying(): void {
    this.order.setState(this.order.getPayment())
  }
  cancelOrder(): void {
    throw new Error('Operation not supported, the order has not yet been paid')
  }
  dispatchOrder(): void {
    throw new Error('Operation not supported, the order has not yet been paid')
  }

}
```

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

```
import { State } from '../interface/state'
import { Order } from './order'

export class Canceled implements State {

  constructor(private order: Order){}

  successWhenPaying(): void {
    throw new Error('Operation not supported, the order is canceled')
  }
  cancelOrder(): void {
    throw new Error('Operation not supported, order already canceled')
  }
  dispatchOrder(): void {
    throw new Error('Operation not supported, the order is canceled')
  }

}
```

```
import { State } from '../interface/state'
import { Order } from './order'

export class Payment implements State {

  constructor(private order: Order){}

  successWhenPaying(): void {
    throw new Error('Operation not supported, the order has already been paid')
  }
  cancelOrder(): void {
    this.order.setState(this.order.getCanceled())
  }
  dispatchOrder(): void {
    this.order.setState(this.order.getSent())
  }

}
```

```
import { State } from '../interface/state'
import { Order } from './order'

export class Sent implements State {

  constructor(private order: Order){}
```

```
  successWhenPaying(): void {
    throw new Error('Operation not supported, the order has already been paid and sent')
  }
  cancelOrder(): void {
    throw new Error('Operation not supported, the order has already been sent')
  }
  dispatchOrder(): void {
    throw new Error('Operation not supported, the order has already been sent')
  }

}
```

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.

```
import { State } from '../interface/state'
import { AwaitingPayment } from './awaiting-payment'
import { Canceled } from './canceled'
import { Payment } from './payment'
import { Sent } from './sent'
export class Order {

  private awaitingPayment: State
  private payment: State
  private canceled: State
  private sent: State
  private state: State

  constructor() {
    this.awaitingPayment = new AwaitingPayment(this)
    this.payment = new Payment(this)
    this.canceled = new Canceled(this)
    this.sent = new Sent(this)

    this.state = this.awaitingPayment
  }

  successWhenPaying() {
    try {
      this.state.successWhenPaying()
    } catch (error) {
      console.error((error as Error).message)
    }
  }

  cancelOrder() {
```

```
    try {
      this.state.cancelOrder()
    } catch (error) {
      console.error((error as Error).message)
    }
  }

  dispatchOrder() {
    try {
      this.state.dispatchOrder()
    } catch (error) {
      console.error((error as Error).message)
    }
  }

  getAwaitingPayment(): State {
    return this.awaitingPayment
  }

  getPayment(): State {
    return this.payment
  }

  getCanceled(): State {
    return this.canceled
  }

  getSent(): State {
    return this.sent
  }

  setState(state: State): void {
    this.state = state
  }

}
```

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.

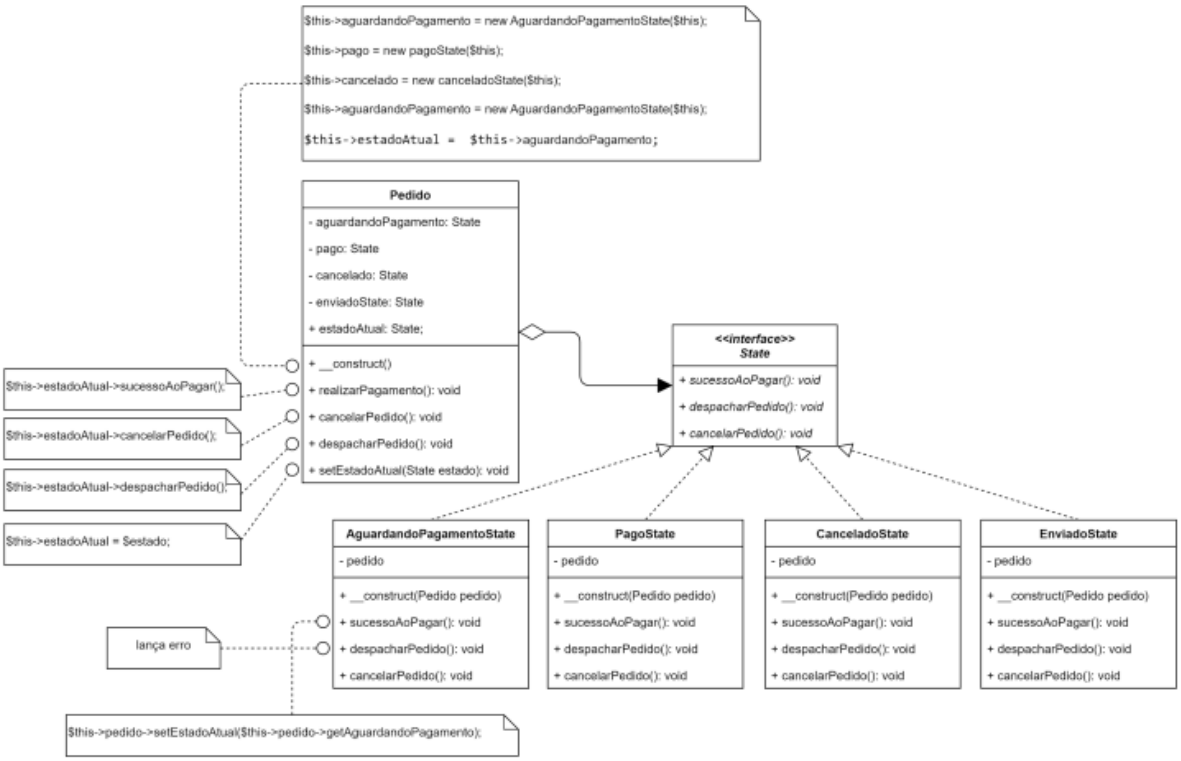


Diagrama de classes do exemplo (Getters foram ocultados)

Aplicabilidade

- Quando 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.

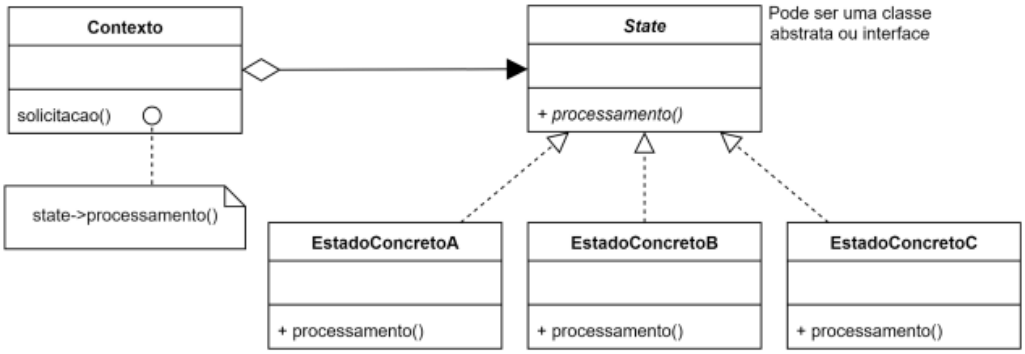


Diagrama de Classes

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.