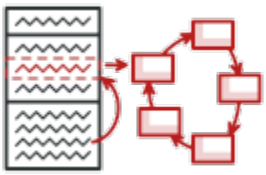
refactoring.guru

State em Go / Padrões de Projeto

8–10 minutes



O **State** é um padrão de projeto comportamental que permite que um objeto altere o comportamento quando seu estado interno for alterado.

O padrão extrai comportamentos relacionados ao estado em classes separadas de estado e força o objeto original a delegar o trabalho para uma instância dessas classes, em vez de agir por conta própria.

Exemplo conceitual

Vamos aplicar o padrão de design State no contexto das máquinas de venda automática. Para simplificar, vamos supor que a máquina de venda automática tenha apenas um tipo de item ou produto. Além disso, para simplificar, vamos supor que uma máquina de venda automática pode estar em 4 estados diferentes:

- hasItem
- noItem

- itemRequested
- hasMoney

  Uma máquina de venda automática também terá ações diferentes. Mais uma vez para simplificar, vamos supor que existam apenas quatro ações:

- Selecionar o item
- Adicionar o item
- Inserir dinheiro
- Dispensar item

  O padrão de design State deve ser usado quando o objeto pode estar em muitos estados diferentes e, dependendo da solicitação de entrada, o objeto precisa alterar seu estado atual.

  Em nosso exemplo, uma máquina de venda automática pode estar em muitos estados diferentes, e esses estados mudarão continuamente de um para outro. Digamos que a máquina de venda automática esteja em `itemRequested`. Uma vez que a ação “Inserir dinheiro” ocorre, a máquina se move para o estado `hasMoney`.

  Dependendo de seu estado atual, a máquina pode se comportar de maneira diferente para as mesmas solicitações. Por exemplo, se um usuário deseja comprar um item, a máquina irá prosseguir se estiver em `hasItemState` ou rejeitará em `noItemState`.

  O código da máquina de venda automática não se polui com essa lógica; todo o código dependente de estado vive nas respectivas implementações de estado.

vendingMachine.go: Contexto

```
package main

import "fmt"

type VendingMachine struct {
    hasItem       State
    itemRequested State
    hasMoney      State
    noItem        State

    currentState State

    itemCount int
    itemPrice int
}

func newVendingMachine(itemCount, itemPrice int)
*VendingMachine {
    v := &VendingMachine{
        itemCount: itemCount,
        itemPrice: itemPrice,
    }
    hasItemState := &HasItemState{
        vendingMachine: v,
    }
    itemRequestedState := &ItemRequestedState{
        vendingMachine: v,
    }
```

```
    hasMoneyState := &HasMoneyState{
        vendingMachine: v,
    }
    noItemState := &NoItemState{
        vendingMachine: v,
    }

    v.setState(hasItemState)
    v.hasItem = hasItemState
    v.itemRequested = itemRequestedState
    v.hasMoney = hasMoneyState
    v.noItem = noItemState
    return v
}

func (v *VendingMachine) requestItem() error {
    return v.currentState.requestItem()
}

func (v *VendingMachine) addItem(count int) error {
    return v.currentState.addItem(count)
}

func (v *VendingMachine) insertMoney(money int) error {
    return v.currentState.insertMoney(money)
}

func (v *VendingMachine) dispenseItem() error {
    return v.currentState.dispenseItem()
}
```

```
func (v *VendingMachine) setState(s State) {
    v.currentState = s
}

func (v *VendingMachine) incrementItemCount(count int) {
    fmt.Printf("Adding %d items\n", count)
    v.itemCount = v.itemCount + count
}
```

state.go: Interface do state

```
package main

type State interface {
    addItem(int) error
    requestItem() error
    insertMoney(money int) error
    dispenseItem() error
}
```

noItemState.go: State concreto

```
package main

import "fmt"

type NoItemState struct {
    vendingMachine *VendingMachine
}
```

```
func (i *NoItemState) requestItem() error {
    return fmt.Errorf("Item out of stock")
}

func (i *NoItemState) addItem(count int) error {
    i.vendingMachine.incrementItemCount(count)
    i.vendingMachine.setState(i.vendingMachine.hasItem)
    return nil
}

func (i *NoItemState) insertMoney(money int) error {
    return fmt.Errorf("Item out of stock")
}
func (i *NoItemState) dispenseItem() error {
    return fmt.Errorf("Item out of stock")
}
```

hasItemState.go: State concreto

```
package main

import "fmt"

type HasItemState struct {
    vendingMachine *VendingMachine
}

func (i *HasItemState) requestItem() error {
    if i.vendingMachine.itemCount == 0 {
```

```
        i.vendingMachine.setState(i.vendingMachine.noItem)
        return fmt.Errorf("No item present")
    }
    fmt.Printf("Item requestd\n")
    i.vendingMachine.setState(i.vendingMachine.itemRequested)
    return nil
}

func (i *HasItemState) addItem(count int) error {
    fmt.Printf("%d items added\n", count)
    i.vendingMachine.incrementItemCount(count)
    return nil
}

func (i *HasItemState) insertMoney(money int) error {
    return fmt.Errorf("Please select item first")
}
func (i *HasItemState) dispenseItem() error {
    return fmt.Errorf("Please select item first")
}
```

itemRequestedState.go: State concreto

```
package main

import "fmt"

type ItemRequestedState struct {
    vendingMachine *VendingMachine
}
```

```
func (i *ItemRequestedState) requestItem() error {
    return fmt.Errorf("Item already requested")
}

func (i *ItemRequestedState) addItem(count int) error {
    return fmt.Errorf("Item Dispense in progress")
}

func (i *ItemRequestedState) insertMoney(money int) error {
    if money < i.vendingMachine.itemPrice {
        return fmt.Errorf("Inserted money is less. Please insert %d",
i.vendingMachine.itemPrice)
    }
    fmt.Println("Money entered is ok")
    i.vendingMachine.setState(i.vendingMachine.hasMoney)
    return nil
}
func (i *ItemRequestedState) dispenseItem() error {
    return fmt.Errorf("Please insert money first")
}
```

hasMoneyState.go: State concreto

```
package main

import "fmt"

type HasMoneyState struct {
    vendingMachine *VendingMachine
```

```
}

func (i *HasMoneyState) requestItem() error {
    return fmt.Errorf("Item dispense in progress")
}

func (i *HasMoneyState) addItem(count int) error {
    return fmt.Errorf("Item dispense in progress")
}

func (i *HasMoneyState) insertMoney(money int) error {
    return fmt.Errorf("Item out of stock")
}
func (i *HasMoneyState) dispenseItem() error {
    fmt.Println("Dispensing Item")
    i.vendingMachine.itemCount = i.vendingMachine.itemCount - 1
    if i.vendingMachine.itemCount == 0 {
        i.vendingMachine.setState(i.vendingMachine.noItem)
    } else {
        i.vendingMachine.setState(i.vendingMachine.hasItem)
    }
    return nil
}
```

main.go: Código cliente

```
package main

import (
    "fmt"
```

```
    "log"
)

func main() {
    vendingMachine := newVendingMachine(1, 10)

    err := vendingMachine.requestItem()
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.insertMoney(10)
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.dispenseItem()
    if err != nil {
        log.Fatalf(err.Error())
    }

    fmt.Println()

    err = vendingMachine.addItem(2)
    if err != nil {
        log.Fatalf(err.Error())
    }

    fmt.Println()
```

```
    err = vendingMachine.requestItem()
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.insertMoney(10)
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.dispenseItem()
    if err != nil {
        log.Fatalf(err.Error())
    }
}
```

output.txt: Resultados da execução

```
Item requestd
Money entered is ok
Dispensing Item

Adding 2 items

Item requestd
Money entered is ok
Dispensing Item
```