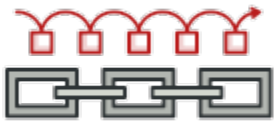
refactoring.guru

Chain of Responsibility em Go

4–6 minutes



O **Chain of Responsibility** é um padrão de projeto comportamental que permite passar a solicitação ao longo da cadeia de handlers em potencial até que um deles lide com a solicitação.

O padrão permite que vários objetos tratem a solicitação sem acoplar a classe remetente às classes concretas dos destinatários. A cadeia pode ser composta dinamicamente em tempo de execução com qualquer handler que siga uma interface de handler padrão.

Exemplo conceitual

Vejamos o padrão Chain of Responsibility com o caso de um aplicativo de hospital. Um hospital pode ter vários departamentos, como:

- Recepção
- Médico

- Sala de remédios
- Caixa

  Sempre que chega algum paciente, ele vai primeiro para a Recepção, depois para o Médico, depois para a Sala de Remédios e depois para o Caixa (e assim por diante). O paciente está sendo enviado por uma cadeia de departamentos, onde cada departamento o envia mais adiante na cadeia, uma vez que sua função esteja concluída.

  O padrão é aplicável quando há vários candidatos para processar a mesma solicitação. Quando você não quer que o cliente escolha o receptor, pois vários objetos podem lidar com a solicitação. Além disso, você deseja desacoplar o cliente dos receptores. O cliente só precisa conhecer o primeiro elemento da cadeia.

  Como no exemplo do hospital, o paciente primeiro vai até a recepção. Então, com base no status atual do paciente, a recepção envia para o próximo handler da cadeia.

  department.go: Interface do handler

```
package main

type Department interface {
    execute(*Patient)
    setNext(Department)
}
```

  reception.go: Handler concreto

```
package main
```

```
import "fmt"

type Reception struct {
    next Department
}

func (r *Reception) execute(p *Patient) {
    if p.registrationDone {
        fmt.Println("Patient registration already done")
        r.next.execute(p)
        return
    }
    fmt.Println("Reception registering patient")
    p.registrationDone = true
    r.next.execute(p)
}

func (r *Reception) setNext(next Department) {
    r.next = next
}
```

doctor.go: Handler concreto

```
package main

import "fmt"

type Doctor struct {
    next Department
```

```
}

func (d *Doctor) execute(p *Patient) {
    if p.doctorCheckUpDone {
        fmt.Println("Doctor checkup already done")
        d.next.execute(p)
        return
    }
    fmt.Println("Doctor checking patient")
    p.doctorCheckUpDone = true
    d.next.execute(p)
}

func (d *Doctor) setNext(next Department) {
    d.next = next
}
```

medical.go: Handler concreto

```
package main

import "fmt"

type Medical struct {
    next Department
}

func (m *Medical) execute(p *Patient) {
    if p.medicineDone {
        fmt.Println("Medicine already given to patient")
```

```
        m.next.execute(p)
        return
    }
    fmt.Println("Medical giving medicine to patient")
    p.medicineDone = true
    m.next.execute(p)
}

func (m *Medical) setNext(next Department) {
    m.next = next
}
```

cashier.go: Handler concreto

```
package main

import "fmt"

type Cashier struct {
    next Department
}

func (c *Cashier) execute(p *Patient) {
    if p.paymentDone {
        fmt.Println("Payment Done")
    }
    fmt.Println("Cashier getting money from patient patient")
}

func (c *Cashier) setNext(next Department) {
```

```
    c.next = next
}
```

patient.go

```
package main

type Patient struct {
    name              string
    registrationDone  bool
    doctorCheckUpDone bool
    medicineDone      bool
    paymentDone       bool
}
```

main.go: Código cliente

```
package main

func main() {

    cashier := &Cashier{}


    medical := &Medical{}
    medical.setNext(cashier)


    doctor := &Doctor{}
    doctor.setNext(medical)
```

```
    reception := &Reception{}
    reception.setNext(doctor)

    patient := &Patient{name: "abc"}

    reception.execute(patient)
}
```

output.txt: Resultados da execução

```
Reception registering patient
Doctor checking patient
Medical giving medicine to patient
Cashier getting money from patient patient
```