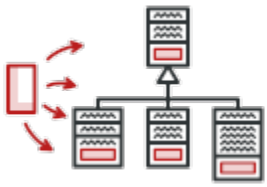
refactoring.guru

Visitor em Go / Padrões de Projeto

6–8 minutes



O **Visitor** é um padrão de projeto comportamental que permite adicionar novos comportamentos à hierarquia de classes existente sem alterar nenhum código existente.

> Leia por que os Visitors não podem ser simplesmente substituídos pela sobrecarga de método em nosso artigo Visitor e Double Dispatch.

Exemplo conceitual

O padrão Visitor permite adicionar comportamento a uma struct sem realmente modificar a struct. Digamos que você seja o mantenedor de uma biblioteca que tem structs de formatos diferentes, como:

- Quadrado
- Círculo
- Triângulo

Cada uma das structs de forma acima implementa a interface de

forma comum.

Depois que as pessoas em sua empresa começaram a usar sua biblioteca incrível, você foi inundado com solicitações de recursos. Vamos revisar um dos mais simples: uma equipe solicitou que você adicionasse o comportamento `getArea` às structs de forma.

Existem muitas opções para resolver este problema.

A primeira opção que vem à mente é adicionar o método `getArea` diretamente na interface de forma e então implementá-lo em cada struct de forma. Esta parece ser a solução certa, mas tem um custo. Como mantenedor da biblioteca, você não quer arriscar quebrar seu precioso código toda vez que alguém solicitar outro comportamento. Ainda assim, você deseja que outras equipes estendam sua biblioteca de alguma forma.

A segunda opção é que a equipe que está solicitando o recurso pode implementar o comportamento por conta própria. Porém, nem sempre isso é possível, pois esse comportamento pode depender do código privado.

A terceira opção é resolver o problema acima usando o padrão Visitor. Começamos definindo uma interface de visitante como esta:

```
type visitor interface {
    visitForSquare(square)
    visitForCircle(circle)
    visitForTriangle(triangle)
}
```

As funções `visitForSquare (square)`, `visitForCircle (circle)`, `visitForTriangle (triangle)` nos permitem

adicionar funcionalidade a quadrados, círculos e triângulos, respectivamente.

Está se perguntando por que não podemos ter um único método `visit(shape)` na interface do visitante? O motivo é que a linguagem Go não oferece suporte à sobrecarga de método, então você não pode ter métodos com os mesmos nomes, mas com parâmetros diferentes.

Agora, a segunda parte importante é adicionar o método `accept` à interface de forma.

```
func accept(v visitor)
```

Todas as structs de forma precisam definir este método, de forma semelhante a este:

```
func (obj *square) accept(v visitor){
   v.visitForSquare(obj)
}
```

Espere um segundo, eu não mencionei que não queremos modificar nossas structs de forma existentes? Infelizmente, sim, ao usar o padrão Visitor, temos que alterar nossas structs de forma. Mas essa modificação só será feita apenas uma vez.

No caso de adicionar qualquer outro comportamento, como `getNumSides`, `getMiddleCoordinates`, usaremos a mesma função `accept(v visitor)` sem quaisquer mudanças adicionais nas structs de forma.

No final, as structs de forma precisam ser modificadas apenas uma vez, e todas as solicitações futuras para comportamentos diferentes podem ser tratadas usando a mesma função de aceitação. Se a equipe solicitar o comportamento `getArea`,

podemos simplesmente definir a implementação concreta da interface do visitante e escrever a lógica de cálculo da área nessa implementação concreta.

shape.go: Elemento

```
package main

type Shape interface {
    getType() string
    accept(Visitor)
}
```

square.go: Elemento concreto

```
package main

type Square struct {
    side int
}

func (s *Square) accept(v Visitor) {
    v.visitForSquare(s)
}

func (s *Square) getType() string {
    return "Square"
}
```

circle.go: Elemento concreto

```
package main

type Circle struct {
    radius int
}

func (c *Circle) accept(v Visitor) {
    v.visitForCircle(c)
}

func (c *Circle) getType() string {
    return "Circle"
}
```

rectangle.go: Elemento concreto

```
package main

type Rectangle struct {
    l int
    b int
}

func (t *Rectangle) accept(v Visitor) {
    v.visitForrectangle(t)
}

func (t *Rectangle) getType() string {
    return "rectangle"
}
```

visitor.go: Visitor

```
package main

type Visitor interface {
    visitForSquare(*Square)
    visitForCircle(*Circle)
    visitForrectangle(*Rectangle)
}
```

areaCalculator.go: Visitante concreto

```
package main

import (
    "fmt"
)

type AreaCalculator struct {
    area int
}

func (a *AreaCalculator) visitForSquare(s *Square) {


    fmt.Println("Calculating area for square")
}

func (a *AreaCalculator) visitForCircle(s *Circle) {
    fmt.Println("Calculating area for circle")
```

```
}
func (a *AreaCalculator) visitForrectangle(s *Rectangle) {
    fmt.Println("Calculating area for rectangle")
}
```

middleCoordinates.go: Visitante concreto

```
package main

import "fmt"

type MiddleCoordinates struct {
    x int
    y int
}

func (a *MiddleCoordinates) visitForSquare(s *Square) {


    fmt.Println("Calculating middle point coordinates for square")
}

func (a *MiddleCoordinates) visitForCircle(c *Circle) {
    fmt.Println("Calculating middle point coordinates for circle")
}
func (a *MiddleCoordinates) visitForrectangle(t *Rectangle) {
    fmt.Println("Calculating middle point coordinates for rectangle")
}
```

main.go: Código cliente

```
package main

import "fmt"

func main() {
    square := &Square{side: 2}
    circle := &Circle{radius: 3}
    rectangle := &Rectangle{l: 2, b: 3}

    areaCalculator := &AreaCalculator{}

    square.accept(areaCalculator)
    circle.accept(areaCalculator)
    rectangle.accept(areaCalculator)

    fmt.Println()
    middleCoordinates := &MiddleCoordinates{}
    square.accept(middleCoordinates)
    circle.accept(middleCoordinates)
    rectangle.accept(middleCoordinates)
}
```

output.txt: Resultados da execução

```
Calculating area for square
Calculating area for circle
Calculating area for rectangle

Calculating middle point coordinates for square
Calculating middle point coordinates for circle
```

Calculating middle point coordinates for rectangle