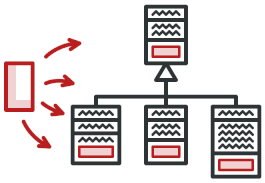

Visitor in Go

Visitor is a behavioral design pattern that allows adding new behaviors to existing class hierarchy without altering any existing code.

Read why Visitors can't be simply replaced with method overloading in our article **Visitor and Double Dispatch**.

Learn more about Visitor →

Navigation

Conceptual Example

The Visitor pattern lets you add behavior to a struct without actually modifying the struct. Let's say you are the maintainer of a lib which has different shape structs such as:

- Square
- Circle
- Triangle

Each of the above shape structs implements the common shape interface.

Once people in your company started to use your awesome lib, you got flooded with feature requests. Let's review one of the simplest ones: a team requested you to add the `getArea` behavior to the shape structs.

There are many options to solve this problem.

The first option that comes to the mind is to add the `getArea` method directly into the shape interface and then implement it in each shape struct. This seems like a go-to solution, but it comes at a cost. As the maintainer of the library, you don't want to risk breaking your precious code each time someone asks for another behavior. Still, you do want other teams to extend

your library somehow.

The second option is that the team requesting the feature can implement the behavior themselves. However, this is not always possible, as this behavior may depend on the private code.

The third option is to solve the above problem using the Visitor pattern. We start by defining a visitor interface like this:

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

The functions `visitForSquare(square)`, `visitForCircle(circle)`, `visitForTriangle(triangle)` will let us add functionality to squares, circles and triangles respectively.

Wondering why can't we have a single method `visit(shape)` in the visitor interface? The reason is that the Go language doesn't support method overloading, so you can't have methods with the same names but different parameters.

Now, the second important part is adding the `accept` method to the shape interface.

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

All of the shape structs need to define this method, similarly to this:

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

Wait a second, didn't I just mention that we don't want to modify our existing shape structs? Unfortunately, yes, when using the Visitor pattern, we do have to alter our shape structs. But this modification will only be done once.

In case adding any other behaviors such as `getNumSides`, `getMiddleCoordinates`, we will use the

same `accept(v visitor)` function without any further changes to the shape structs.

In the end, the shape structs just need to be modified once, and all future requests for different behaviors could be handled using the same accept function. If the team requests the `getArea` behavior, we can simply define the concrete implementation of the visitor interface and write the area calculation logic in that concrete implementation.

shape.go: Element

```
package main

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

square.go: Concrete element

```
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: Concrete element

```
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: Concrete element

```
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: Concrete visitor

```go
package main

import (
    "fmt"
)

type AreaCalculator struct {
    area int
}

func (a *AreaCalculator) visitForSquare(s *Square) {
    // Calculate area for square.
    // Then assign in to the area instance variable.
    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: Concrete visitor

```go
package main

import "fmt"

type MiddleCoordinates struct {
    x int
    y int
}

func (a *MiddleCoordinates) visitForSquare(s *Square) {
    // Calculate middle point coordinates for square.
    // Then assign in to the x and y instance variable.
    fmt.Println("Calculating middle point coordinates for square")
}

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

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

main.go: Client code

```go
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: Execution result

```
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
```

RETURN

READ NEXT

← Template Method in Go

Design Patterns in Java →

Visitor in Other Languages