

## **\*\* INTERFACES IN GO \*\***

- Problems with the structs :-

Ex:-1 Mocking the payment

req1:

- \* we need a data structure for this
- \* so we are making a struct payment

```
type payment struct{}

func (p payment) makePayment(amount float32) {
    razorpayPaymentGw := razorpay{}
    razorpayPaymentGw.pay(amount)
}
```

req2: we need a payment gateway so here we are

```
type razorpay struct{}

func (r razorpay) pay(amount float32) {
    // logic to make payment
    fmt.Println("making payment using razorpay", amount)
}
```

execution1 :

to make the payment we need an instance of payment struct

```
func main() {
    newPayment := payment{}
    newPayment.makePayment(100)
}
```

till now there is not problem everything is working fine

```
•→ golang-course go run 17_interfaces/interface.go
   making payment using razorpay 100
○→ golang-course
```

---

? PROBLEM arises in real world scenarios like

case1: suppose if you have to add another payment gateway like stripe then :-

p1 : we need to make one more struct like razorpay

```
type stripe struct{}

func (s stripe) pay(amount float32) {
    fmt.Println("making payment using stripe", amount)
}
```

p2 : we need to change our makePayment method of payment struct

```
func (p payment) makePayment(amount float32) {
    // razorpayPaymentGw := razorpay{}
    stripePaymentGw := stripe{}
    // razorpayPaymentGw.pay(amount)
    stripePaymentGw.pay(amount)
}
```

execution no change

-----  
PROBLEM

- \* violating the principle of open - closed [solid principle]
- \* as it should be open for extend and close for modification
- \* but here we are modifying existing implementation
- \* and also you can see tight coupling between the method of payment struct and instance of stripe/razorpay struct

-----

Improvement 1:

in the makePayment method you can see on each call we are creating the new instance of [stripe/razorpay]

\* to improve this we will keep the dependency in the struct itself instead of method

```
type payment struct {
    gateway stripe
}
```

and in the method we will not create the instance as we have access of the struct [payment] so we will use the method directly

```
// Open close principle
func (p payment) makePayment(amount float32) {
    // razorpayPaymentGw := razorpay{}
    // stripePaymentGw := stripe{}
    // razorpayPaymentGw.pay(amount)
    p.gateway.pay(amount)
}
```

here we have gateway so we will use the pay method directly

**Problem : there is a problem : till now we haven't used the stripe otherwise we will get error**

```
func main() {
    newPayment := payment{}
    newPayment.makePayment(100)
}
```

\*\* because in the payment {} we must have to pass the payment gateway as defined in the struct

\*\* but previously we have seen not necessary to pass the fields then why ?? this is because :-

Methods, on the other hand, **require that the fields be set properly** when they are called. In the case of your `makePayment` method, it directly accesses the `gateway` field of the `payment` struct. If the `gateway` is not initialized, it would cause an error when trying to invoke

```
p.gateway.pay(amount) .
```

For example, if you call `newPayment.makePayment(100)` on an uninitialized or improperly initialized struct, you will run into issues, because the `gateway` field needs to be set for the method to work properly.

In this case:

```
go Copy code

func (p payment) makePayment(amount float32) {
    p.gateway.pay(amount) // We access `p.gateway`, so it must be initialized
}
```

If the `gateway` field isn't set (i.e., if it's `nil` or the zero value of the type), the method will fail. Therefore, **you need to ensure the struct is properly initialized with all required fields** before calling the method.

\* so its necessary to make the initialize the payment gateway otherwise we will get error

\* so we need to do like this

```
func main() {  
    stripePaymentGw := stripe{}  
    newPayment := payment{  
        gateway: stripePaymentGw,  
    }  
    newPayment.makePayment(100)  
}
```

\* so those parameters of struct which are getting used in the functions attatched to the struct must be initialized otherwise we will get error

-----

Still other problems we have left as if you want to add another payment gateway still we need to change the gateway of struct this is how ??

```
type payment struct {  
    gateway razorpay  
}  
  
func main() {  
    // stripePaymentGw := stripe{}  
    razorpayPaymentGw := razorpay{}  
    newPayment := payment{  
        gateway: razorpayPaymentGw,  
    }  
    newPayment.makePayment(100)  
}
```

and also there is one more problem

- suppose if you have to test the makePayment method then in that case you must have to pass the actual payment gateway like here razorpay

```
type payment struct {  
    gateway razorpay  
}
```

which means you are bound to pass the razorpay only [you need a razorpay instance only]

```
razorpayPaymentGw := razorpay{}
```

```

type fakepayment struct{}

func (f fakepayment) pay(amount float32) {
    fmt.Println("making payment using fake gateway for testing purpose")
}

func main() {
    // stripePaymentGw := stripe{}
    // razorpayPaymentGw := razorpay{}
    fakeGw = fakepayment{}
    newPayment := payment{
        gateway: fakeGw,
    }
}

```

\* here you can see we can't pass the fake payment gateway as we have defined the razorpay gateway in the payment struct

- \* so testing is complex
- \* solution is interface

## INTERFACE

- interfaces are contracts
- contract : means whatever [function in interfaces of go] you mention in the interface that must be defined in the struct if a struct implements that
- you use er or r in the last of the interface as a convention

1. define interface

```

type payer interface {
    pay(amount float32)
}

type payer interface {
    pay(amount float32) bool
}

```

- if it returns something then you can also add in this way

2. and the gateway inside the struct should not be a concrete implementation of any struct or gateway [like stripe or razorpay]

```

type payment struct {
    gateway razorpay
}

```

- \* as this razorpay is concrete implementation of the razorpay

```

type razorpay struct{}

func (r razorpay) pay(amount float32) {
    // logic to make payment
    fmt.Println("making payment using razorpay", amount)
}

```

- same it shouldn't be concrete implementation of any gateway[stripe, razorpay or fake]
- \* so in struct we don't have to use concrete implementation of any gateway
- \*\* here we have to pass the interface like payer

```

type payer interface {
    pay(amount float32)
}

type payment struct {
    gateway payer
}

func (p payment) makePayment(amount float32) {
    p.gateway.pay(amount)
}

```

- \* below you see the razor pay struct and implementation

```

type razorpay struct{}

func (r razorpay) pay(amount float32) {
    //logic to make payment
    fmt.Println("making payment using razorpay", amount)
}

```

- \* if you carefully see pay method of razorpay have same signature [name, arguments, return type] of pay method of payer interface
- \* that means it razorpay struct is implementing the payer interface

**NOTE :** in go we don't need to explicitly mention like other languages where you write a class implements a interface  
In go if a struct method have same signature of a interface then internally it is implementing that interface

- \* now things are fine now you can do testing also it is that simple



\* just make the struct and define the same method of same signature

```
type fakepayment struct{}

func (f fakepayment) pay(amount float32) {
    fmt.Println("making payment using fake gateway for testing purpose")
}

func main() {
    // stripePaymentGw := stripe{}
    // razorpayPaymentGw := razorpay{}
    fakeGw := fakepayment{}
    newPayment := payment{
        gateway: fakeGw,
    }
    newPayment.makePayment(100)
}
```

\* now there is no problem here

\* Now our problem is solved

-----

\* suppose in future if you want to add paypal then this is how you can do that

```
type paypal struct{}

func (p paypal) pay(amount float32) {
    fmt.Println("making payment using paypal", amount)
}

paypalGw := paypal{}
newPayment := payment{
    gateway: paypalGw,
}
newPayment.makePayment(100)
```

always added bet  
appended. It retur  
any write error en

\* no problem is here to add a new as we are extending not modifying

\* if you want to add more methods then also you can do that

```
type payer interface {
    pay(amount float32)
    refund(amount float32, account string)
}
```

\*\*\*\*\* most important : keep in mind \*\*\*\*\*

```
paypalGw := paypal{}  
newPayment := payment{  
    gateway: paypalGw,  
}  
newPayment.makePayment(100)
```

- here now it can't use this paypal payment gateway
- because now paypal struct is not implementing the interface payer

\*\* whether it is using that method [refund] or not if it have to use any method of that interface it must have to implement that interface and for implementing that interface it have to define all the methods in the interface with same signature [no matter whether you are using all the methods or not but you have to define that]

```
type paypal struct{  
  
func (p paypal) pay(amount float32) {  
    fmt.Println("making payment using paypal", amount)  
}  
  
func (p paypal) refund(amount float32, account string) {  
    |  
}
```

-----  
structs and interfaces works implicitly together in go  
-----  
dependency inversion here we are doing

## Dependency Inversion Principle (DIP):

The **Dependency Inversion Principle** is one of the SOLID principles that aims to:

1. **High-level modules** should not depend on **low-level modules**. Both should depend on **abstractions** (interfaces).
2. **Abstractions** should not depend on **details**. The **details** should depend on **abstractions**.

In simpler terms, the high-level logic (e.g., your `payment` struct) should not depend on specific implementations (like `razorpay`, `stripe`, or `fake`). Instead, both the high-level logic and low-level details should depend on a common interface (an abstraction). This allows for easier swapping of implementations without modifying the high-level code.



- **Dependency Inversion in Go** is demonstrated by the `payment` struct depending on the `paymenter` interface rather than on any concrete payment gateway (like `razorpay`, `stripe`, or `paypal`).
- The concrete payment gateways (e.g., `razorpay`, `paypal`) implement the `paymenter` interface, which defines the contract.
- The `payment` struct uses the `paymenter` interface as a dependency, ensuring that the high-level logic of making a payment doesn't depend on the specific implementation details of the payment method. This makes the system flexible and open to extension without modifying the existing code.

This design allows you to add new payment methods (like `stripe`, `square`, etc.) without changing the high-level logic of the `payment` struct, adhering to both the **Open/Closed Principle** and the **Dependency Inversion Principle**.

---

## How It Applies to Your Code:


In the code you provided, **Dependency Inversion** is demonstrated in the following ways:

1. `paymenter` Interface:

- The `paymenter` interface is an abstraction that defines the contract for any payment gateway. This contract specifies that any type that implements this interface must have a `pay` method, and optionally a `refund` method.

```
go

type paymenter interface {
    pay(amount float32)
    refund(amount float32, account string)
}
```


 Copy code

This interface is the **abstraction** that the high-level `payment` struct depends on, rather than concrete implementations of payment gateways like `razorpay`, `stripe`, or `paypal`.

## 2. `payment` Struct Dependent on `paymentner` Interface:

- The `payment` struct does **not** depend on any concrete payment gateway (like `razorpay` or `paypal`). Instead, it holds a reference to the `paymentner` interface, which means it can work with any payment gateway that implements the `paymentner` interface.

go

 Copy code


```
type payment struct {  
    gateway paymentner // payment depends on the abstraction (paymentner interface)  
}
```

The `payment` struct does not care whether the gateway is `paypal`, `razorpay`, `stripe`, or any other payment method. It only depends on the **abstract contract** defined by the `paymentner` interface.

## 3. Low-Level Implementations of Payment Gateways (`razorpay`, `paypal`, etc.):

- The concrete payment gateways (like `razorpay`, `paypal`, `fake`) implement the `paymentner` interface. The `razorpay`, `paypal`, and `fake` types all provide their own versions of the `pay` and `refund` methods.

go

 Copy code

```
type razorpay struct{}  
func (r razorpay) pay(amount float32) {  
    fmt.Println("making payment using razorpay", amount)  
}  
  
type paypal struct{}  
func (p paypal) pay(amount float32) {  
    fmt.Println("making payment using paypal", amount)  
}  
func (p paypal) refund(amount float32, account string) {  
    fmt.Println("we are processing refund for paypal", amount, account)  
}
```

