

Go interface segregation redux

Redowan Delowar

Software Engineer, Core Infra, Wolt

Find the slides



Outline

- Interface 101
- Machine representation
- Decoupling with interface
- The bigger the interface, the weaker the abstraction
- Tiny interfaces in the stdlib
- Composing interfaces
- Interface segregation
- Example

Interfaces are for abstraction

An interface is a **contract** with the outside world, your best face forward.

— Anonymous

Go interface

*An **interface type** is defined as a **set of method signatures**.*

*A **value of interface type** can hold any **value** that **implements** those **methods**.*

— Tour of Go

Go interface (cont')

```
type Speaker interface {  
    Speak() string  
}
```

```
type Dog struct{}  
func (Dog) Speak() string { return "Woof!" }
```

```
type Cat struct{}  
func (Cat) Speak() string { return "Meow!" }
```

```
func main() {  
    animals := []Speaker{Dog{}, Cat{}}  
    for _, a := range animals {  
        fmt.Println(a.Speak())  
    }  
}
```

Output:

Woof!

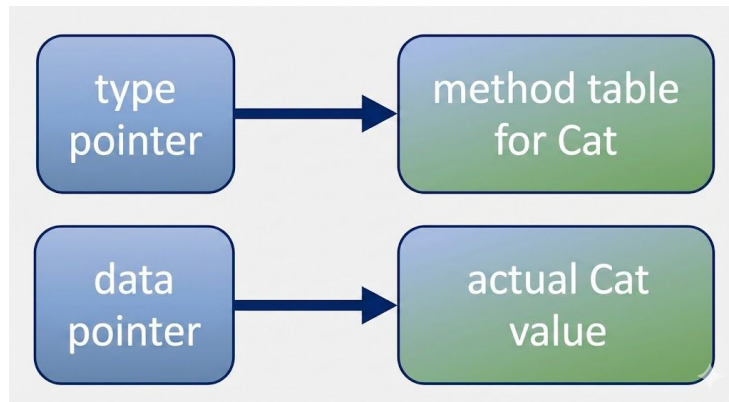
Meow!

Machine representation

When you assign a value to an interface like this:

```
var cat Speaker = Cat{}
```

the interface value occupies two machine words

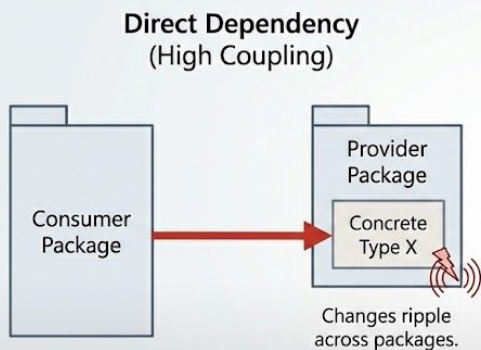


One word holds the concrete type information

The other holds a pointer to the underlying data

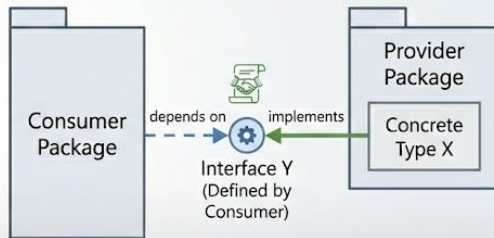
Use interfaces to decouple packages

Decoupling with Interfaces



Changes ripple across packages.

Interface as Contract (Loose Coupling)



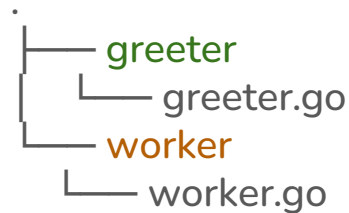
Contract isolates changes. Fewer ripples. ✨

- Interfaces break direct dependencies on concrete types.
- Interfaces act as small contracts between packages.
- Less coupling means fewer ripple effects across packages.
- The consuming package defines the interface.

Concrete dependency

- There are two packages: **greeter** and **worker**
- **greeter** has a concrete type that **worker** imports & uses
- When the type in **greeter** changes, **worker** breaks - tight coupling
- This setup makes refactoring and testing harder

Package structure



Concrete dependency (cont')

greeter/greeter.go

```
type Greeter struct{}
```

```
func (Greeter) Greet(name string) string {  
    return "Hello, " + name  
}
```

worker/worker.go

```
package worker
```

```
import "example.com/project/greeter"
```

```
func Do(g greeter.Greeter) string {  
    return g.Greet("gopher")  
}
```

Concrete dependency makes testing tricky

`worker/worker_test.go`

```
package worker
```

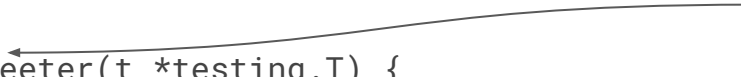
```
import (  
    "testing"  
    "example.com/project/greeter"  
)
```

`worker` test depends on
`greeter`



```
func TestDo_UsesRealGreeter(t *testing.T) {  
    g := greeter.Greeter{}  
    got := Do(g)  
    want := "Hello, gopher"  
    ...  
}
```

`Do` accepts exactly
`greeter.Greeter` and
nothing else

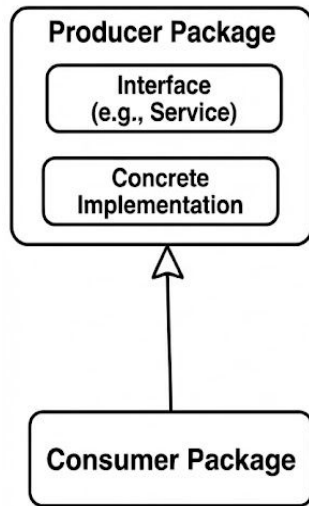


Three kinds of interface

1. Producer-side interface
2. Consumer-side interface
3. Interface in a shared package

Producer side interface

- Producer package provides the interface
- It also implements the type that satisfies the contract
- Consumer package depends on the producer-side interface



Producer-side interface

Producer-side interface (cont')

greeter/greeter.go (producer)

```
package greeter

// Producer exposes the interface
type Greeter interface {
    Greet(name string) string
}

// Private concrete implementation
type greeter struct{}

func (greeter) Greet(name string) string {
    return "Hello, " + name
}
```

worker/worker.go (consumer)

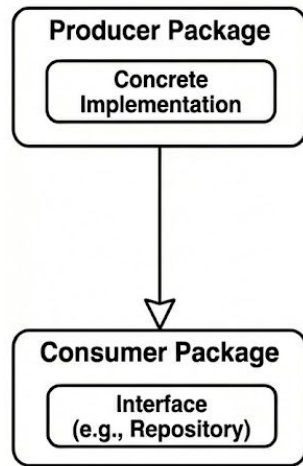
```
package worker

import "example.com/project/greeter"

func Do(g greeter.Greeter) string {
    return g.Greet("gopher")
}
```

Consumer-side interface

- Producer package only provides the concrete type
- Consumer package defines the contract it needs and only depends on that
- The concrete type from the producer package is injected



Consumer-side interface

Consumer-side interface (cont')

greeter/greeter.go (producer)

```
package greeter
```

```
// Producer only provides the concrete type
```

```
type greeter struct{}
```

```
func (greeter) Greet(name string) string {  
    return "Hello, " + name }  
}
```

worker/worker.go (consumer)

```
package worker
```

```
// Consumer defines what it needs.
```

```
type Greeter interface {
```

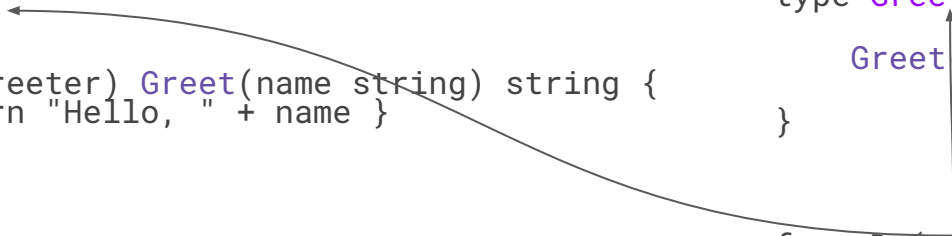
```
    Greet(name string) string
```

```
}
```

```
func Do(g Greeter) string {
```

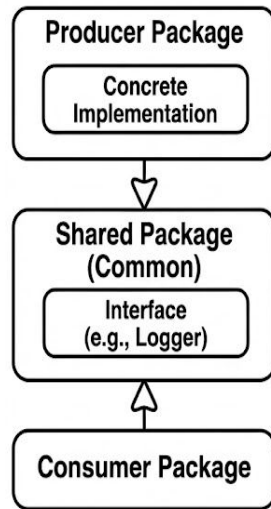
```
    return g.Greet("gopher")
```

```
}
```



Shared interface

- Interface lives in a separate third package
- Producer package provides a type that implicitly implements the contract
- Consumer depends on the shared interface and injects the concrete type



Interface in a shared package

Shared interface (cont')

greetapi/greetapi.go (shared package)

```
package greetapi

type Greeter interface {
    Greet(name string) string
}
```

greeter/greeter.go (producer)

```
package greeter

type greeter struct{}

func (greeter) Greet(name string) string {
    return "Hello, " + name }
```

worker/worker.go (consumer)

```
package worker

import "example.com/project/greetapi"

func Do(g greetapi.Greeter) string {
    return g.Greet("gopher")
}
```

Testing wins

- The test can inject a **fake** that implements the interface
- No need to depend on the real **Greeter** implementation

`worker/worker_test.go`

```
package worker
```

```
import "testing"
```

```
// Fake implementation for testing.
```

```
type fakeGreeter struct{}
```

```
func (fakeGreeter) Greet(name string) string {  
    return "Hi, " + name  
}
```

```
func TestDo(t *testing.T) {
```

```
    g := fakeGreeter{}
```

```
    got := Do(g)
```

```
    want := ...
```

```
}
```

Which interface type to pick

- Producer-side interfaces are more common in libs, e.g. `io.Reader`, `io.Writer`
- Refactoring a producer-side interface will break the dependents
- In applications, start with consumer-side ones
- These are much easier to change
- Centralize your interfaces in a shared package

Tiny interfaces in the stdlib

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```

```
type Writer interface {  
    Write(p []byte) (n int, err error)  
}
```

```
type ReadWriter interface {  
    Reader  
    Writer  
}
```

Interface segregation

Clients should not be forced to depend on methods they do not use.

— Robert C. Martin (SOLID, interface segregation principle)

Interface segregation (cont')

[Your]

code shouldn't accept anything it doesn't use

function should only accept an interface when it needs all of its methods

function shouldn't accept a narrow interface when a broader one will do

function shouldn't accept an interface and then use only a subset of its methods

Interface segregation - example

```
type FileStorage struct{}

func (FileStorage) Save([]byte) error { return nil }

func (FileStorage) Load(string) ([]byte, error) { return nil, nil }

func Backup(fs FileStorage, data []byte) error {
    return fs.Save(data)
}
```

Concrete type, hard to test

Depend on behavior

```
type Storage interface {  
    Save([]byte) error  
    Load(string) ([]byte, error)  
}
```

```
func Backup(s Storage, data []byte) error {  
    return s.Save(data)  
}
```

Better, but still too broad.

Narrow it down

```
type Saver interface {  
    Save([]byte) error  
}  
  
func Backup(s Saver, data []byte) error {  
    return s.Save(data)  
}
```

Now Backup depends only on what it needs.

Testing is trivial now

```
type fakeStorage struct {  
    saved []byte  
    err    error  
}  
  
func (f *fakeStorage) Save(b []byte) error {  
    f.saved = b  
    return f.err  
}  
  
func TestBackup(t *testing.T) {  
    fs := &fakeStorage{}  
    data := []byte("hello")  
  
    err := Backup(fs, data)  
    ...  
}
```



```
type Saver interface {  
    Save([]byte) error  
}
```

Where to put that interface

Interfaces belong in the package that uses them.

— Go Code Review Comments

Keep them close to the consumer

Let the producer return concrete types

Rule of thumb

Insert a seam between two tightly coupled components
by placing a consumer-side interface that exposes
only the methods the caller invokes

References

- [Revisiting interface segregation](#)
- [Err on smaller interfaces](#)
- [Consumer-defined contracts](#)
- [Interface upgrades](#)
- [Interface smuggling](#)

Thank you!