INTERFEJSY

Pakiet `draw` rysujący obiekty na ekranie

```
package draw

type Screen struct {
}

func (s *Screen) DrawRect(x, y, width, height int) {}

func (s *Screen) DrawCircle(x, y, r int) {}
```

Wykorzystanie i testowanie pakietu `draw`

```go
package engine
import "example/draw"

func Refresh(s *draw.Screen) {
    // ...
    s.DrawRect(100,100,25,10)
}
```

- W teście jednostkowym jesteśmy zmuszeni wykorzystać typ draw.Screen
- Nie chcemy testować pakietu `draw` tylko metodę `Refresh`
- Chcemy aby metoda `Refresh` przyjęła w parametrze typ, nad którego implementacją mamy kontrolę podczas testu

Duck Typing

Duck typing – rozpoznawanie typu obiektu nie na podstawie deklaracji, ale przez badanie metod udostępnionych przez obiekt. Technika ta wywodzi się z powiedzenia: „jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką”.

```
type RoboDuck struct {}
func (d *RoboDuck) Quack() {}
func (d *RoboDuck) Walk() {}
```

```
type Duck struct {}
func (d *Duck) Quack() {}
func (d *Duck) Walk() {}
```

Interfejs

- Interfejs składa się z sygnatur metod
- Sygnatura to nazwa metody, lista parametrów wejściowych i typów wyjściowych (o ile występują)

```
type Drawer interface {
    DrawRect(int, int, int, int)
    DrawCircle(int, int, int)
}
```

Interfejs

- Uwaga! To jest antypattern!
- Interfejs powinien zostać zdefiniowany po stronie “klienta”, czyli w pakiecie `engine`

```
package draw

type Drawer interface {
    DrawRect(int, int, int, int)
    DrawCircle(int, int, int)
}

type Screen struct {
}

func (s *Screen) DrawRect(x, y, width, height int) {}
func (s *Screen) DrawCircle(x, y, r int) {}
```

```
package engine
import "example/draw"

func Refresh(s draw.Drawer) {
    s.DrawRect(100,100,25,10)
}
```

Interfejs

Dlaczego definicja interfejsu powinna znajdować się po stronie “klienta”?

- Zmiana pakietu `draw` na inny, który nie zawiera metody `DrawCircle`, wymusza zmiany we wszystkich metodach korzystających z `draw.Drawer`
- Importując interfejs `draw.Drawer` nie mamy wpływu na zawartość interfejsu (listę sygnatur metod)

Jak to zrobić lepiej?

```
package engine
import "example/draw"

type Drawer interface {
    DrawRect(int, int, int, int)
    DrawCircle(int, int, int)
}

func Refresh(s Drawer) {
    s.DrawRect(100,100,25,10)
}
```

Nie trzeba korzystać ze wszystkich metod!

```go
package engine
import "example/draw"

// type Drawer interface {
//     DrawRect(int, int, int, int)
//     DrawCircle(int, int, int)
// }

type Rectangler interface {
    DrawRect(int, int, int, int)
}

func Refresh(r Rectangler) {
    r.DrawRect(100,100,25,10)
}
```

Pusty interface (any)

- Szczególnym przypadkiem jest typ `interface{}` lub jego alias `any`
- Dowolny typ spełnia interfejs bez metod jednak nie powinien być nadużywany

Best Practices

- Przyjmuj interfejsy, zwracaj struktury
- Interfejs spełnia jedną logikę związaną z jego nazwą
- Nazwa interfejsu zazwyczaj jest rzeczownikiem kończącym się na -er

Przykłady interfejsów w bibliotece standardowej Go

```go
type error interface {
    Error() string
}
```

```go
package main

import "fmt"

type myErr struct {
    msg  string
    code int
}

func (e *myErr) Error() string {
    return fmt.Sprintf("%s, code %d", e.msg, e.code)
}
```

```go
func myFunc() *myErr {
    return &myErr{
        msg:  "not found",
        code: 404,
    }
}

func myService() error {
   e := myFunc()
   return fmt.Errorf("myFunc returned error: %v",e)
}

func main() {
    if err := myService(); err != nil {
        fmt.Println(err)
        return
    }
}

// output:
// myFunc returned error: not found, code 404
```

Przykłady interfejsów w bibliotece standardowej Go

Pakiet fmt

```
type Stringer interface {
    String() string
}
```

```
package main
import "fmt"

type Box struct {
    x, y, width, height int
}

// func (b *Box) String() string {
//     return fmt.Sprintf("X: %d, Y: %d, Width: %d, Height: %d",
//                        b.x, b.y, b.width, b.height)
//}

func main() {
        b := new(Box)
        fmt.Println(b)
}

// output:
// &{0 0 0 0}

// output z metodą String()
// X: 0, Y: 0, Width: 0, Height: 0
```

Zapoznaj się z innymi interfejsami w stdlib

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

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

type ReadCloser interface {
    Reader
    Closer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```