INTRODUCCIÓN A LA CLASE

GO TESTING

Objetivos de esta clase

- Conocer y entender los Test Doubles.
- Comprender los tipos de Test Doubles.
- Implementar Test Doubles con Go.
- Saber cuándo aplicar cada tipo.

TEST DOUBLE

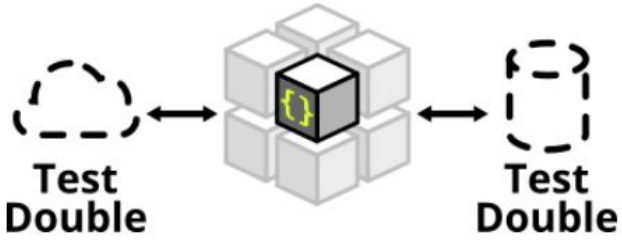
GO TESTING

// ¿Qué es Test Double?

Es necesario para simular o emular alguna otra dependencia usada por el objeto de prueba.

Entendiendo el Test Double

Se les llama Test Double, para hacer referencia al uso de “Dobles” en la filmación de películas o afines. Consiste en emplear “reemplazos” a objetos requeridos por el código que queremos probar.



TIPOS DE TEST DOUBLE

GO TESTING

Tipos de Test Double

Según el objetivo del test y su comportamiento, los distintos tipos de Test Double son:

- Dummy
- Stub
- Spy
- Mock
- Fake

DUMMY TEST

GO TESTING

Dummy

Es un objeto vacío que implementa una interfaz específica. Su uso, comportamiento o respuesta es irrelevante y no nos importa. Sólo lo usamos para satisfacer dependencias necesarias para la ejecución del código que estamos probando. Por ejemplo, supongamos que una función Sumar() requiere de un objeto logger para efectos de trazabilidad. La trazabilidad no nos importa, porque solo queremos probar el método. Pero necesitamos crear un logger dummy que realmente no haga nada, pero que nos permita ejecutar la prueba.

Dummy

Para probar esta función necesitaremos un Dummy del tipo logger, no nos importa que hace pero es requerido por la función.

```go
package calculadora

type Logger interface {
    Log(string) error
}

// Función que recibe dos enteros, un objeto del tipo logger y retorna la   suma resultante
func Sumar(num1, num2 int, logger Logger) int {
    err := logger.Log("Ingreso a Función Sumar")
    if err != nil {
        return -99999
    }
    return num1 + num2
}
```

{}

```go
// se crea un un struct dummyLogger
type dummyLogger struct{}

//  Se escriben las funciones necesarios para que dummyLogger cumpla con la interfaz que va a reemplazar (Logger)
func (d *dummyLogger) Log(string) error {
    return nil
}

func TestSumar(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := 8
    // Se genera el objeto dummy a usar para satisfacer la necesidad de la función Sumar
    myDummy := &dummyLogger{}
    // Se ejecuta el test
    resultado := Sumar(num1, num2, myDummy)
    // Se validan los resultados aprovechando testify
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")
}
```

Dummy

Para el test se creó un dummy Logger que básicamente no tiene ningún uso, salvo satisfacer la necesidad de la Función Suma.

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    go-testing/calc  0.295s
```

STUB TEST

GO TESTING

Stub

El propósito de un Stub es el de proveer valores concretos para guiar al test en una determinada dirección. Implementa métodos y devuelve valores requeridos para el test. Por ejemplo: en la función “Sumar” - descrita previamente - existe una condición, en la que si el Logger.Log, retorna un error, la suma no se ejecuta sino que retorna el valor -99999.

Para probar esta condición no es suficiente con crear un Dummy, sino que es necesario que el objeto que estamos simulando devuelve específicamente un error. Esto es precisamente lo que hace el Stub.

{}

```go
// se crea un un struct stubLogger
type stubLogger struct{}

//  Se escribe las funciones necesarias para que stubLogger retorne exactamente lo que necesitamos
func (s *stubLogger) Log(string) error {
    return errors.New("error desde stub")
}

func TestSumarError(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := -99999
    // Se genera el objeto stub a usar para satisfacer la necesidad de la función Sumar
    myStub := &stubLogger{}
    // Se ejecuta el test
    resultado := Sumar(num1, num2, myStub)
    // Se validan los resultados aprovechando testify
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")
}
```

Stub

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    go-testing/calc  0.145s
```

SPY TEST

GO TESTING

Spy

En ocasiones es necesario comprobar o asegurarse de haber llamado a un método para dar el test como válido. Para esto utilizamos un Spy. Y la comprobación consiste en consultarle al Spy si el método en cuestión, fue invocado o utilizado durante la ejecución. De allí el nombre de este tipo de tests, es un espía que nos informa cuando algo sucede. Nuestro próximo test consiste en comprobar que efectivamente el método Log del objeto logger sea invocado durante la prueba.

{}

```go
// se crea un un struct spy compuesto por un booleano que nos informará si ocurre el
llamado a Log
type spyLogger struct {
    spyCalled bool
}
//  Para espiar creamos un loggerSpy que setea en true spyCalled si entra al método
func (s *spyLogger) Log(string) error {
    s.spyCalled = true
    return nil
}
func TestSumarConSpy(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := 8
    // Se genera el objeto spy a usar
    mySpy := &spyLogger{}
    // Se ejecuta el test y se validan el resultado y que spyCalled sea true para dar el
test por válido
    resultado := Sumar(num1, num2, mySpy)
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")
    assert.True(t, mySpy.spyCalled)
}
```

Spy

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    go-testing/calc  1.085s
```

MOCK TEST

GO TESTING

Mock

El Mock, contrario a un Stub, no es aplicado para devolver valores exactos sino para comprobar todo el funcionamiento interno del método o código que se está probando. El Mock está más interesado en que métodos se han invocado, con que argumentos, cuando y con qué frecuencia.

Un mock siempre es un espía y conoce lo que se se está testeando. Y las comprobaciones del test se aplican sobre el mock. Supongamos que ahora tenemos una función Sumar más compleja que llamaremos SumarRestricted(), en la que sólo se devolverá el resultado, si el proceso lo invoca un cliente autorizado para tal fin.

Por lo que recibe por parámetros, el nombre del cliente que está ejecutando el proceso, y la interfaz que valida si dicho cliente está autorizado.

Mock

Para probar esta función necesitaremos un Mock que compruebe que la validación SumaEnabled() está siendo invocada correctamente y que además reciba el cliente correcto.

{}

```go
package calculadora

type Config interface {
    SumaEnabled(cliente string) bool
}

// Función que recibe dos enteros y retorna la suma resultante
func SumarRestricted(num1, num2 int, config Config, cliente string) int {
    if !config.SumaEnabled(cliente) {
        return -99999
    }
    return num1 + num2

}
```

{}

```go
// se crea un un struct mockConfig
type mockConfig struct {
    clienteUsado string
}
// El mock debe implementar el método necesario y comprobar que SumaEnabled sea llamado y que se haga exactamente con el mismo cliente que recibió SumarRestricted
func (m *mockConfig) SumaEnabled(cliente string) bool {
    m.clienteUsado = cliente
    return true
}
func TestSumarRestricted(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    cliente := "John Doe"
    resultadoEsperado := 8
    // Se genera el objeto dummy a usar para satisfacer la necesidad de la función Sumar
    myMock := &mockConfig{}
    // Se ejecuta el test y se valida el resultado y que el mock haya registrado la información correcta
    resultado := SumarRestricted(num1, num2, myMock, cliente)
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")
    assert.Equal(t, cliente, myMock.clienteUsado)
}
```

Mock

Como se indicó previamente, la información del Mock es validada en los “assertions” del test. Por esto el mock nos ayuda a comprobar el funcionamiento correcto del método. Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    go-testing/calc  0.295s
```

FAKE TEST

GO TESTING

Fake

Los Fake son objetos que contienen cierta lógica de negocio adentro. Es una especie de “simulador” que debe generar respuestas distintas de acuerdo a distintos escenarios. Esto permite comprobar validaciones o comportamientos asociados al negocio, en los que además no podemos usar datos o escenarios productivos. Un Fake se distingue del resto de los test Double, ya que ningún otro contiene lógica de negocio. Son tests que tienden a crecer en complejidad en la medida que más lógica contengan.

Fake

Tomando como objeto de pruebas nuevamente la función SumarRestricted, vamos a generar un Fake de la interfaz Config, cuyo método SumaEnabled devuelve “true” para un cliente específico. En otras palabras, queremos que solo un cliente pueda acceder al método Suma, el resto, debe recibir como resultado el número -99999.

{}

```go
package calculadora

type Config interface {
    SumaEnabled(cliente string) bool
}

// Función que recibe dos enteros y retorna la suma resultante
func SumarRestricted(num1, num2 int, config Config, cliente string) int {
    if !config.SumaEnabled(cliente) {
        return -99999
    }
    return num1 + num2
}
```

{}

```go
// se crea un un struct fakeConfig que implemente una lógica en la que sólo habilita la suma al cliente "John Doe"
type fakeConfig struct{}
func (f *fakeConfig) SumaEnabled(cliente string) bool {
    return cliente == "John Doe"
}
func TestSumarRestrictedFake(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    cliente := "John Doe"
    cliente_dos := "Mister Pmosh"
    resultadoEsperado := 8
    resultadoEsperadoError := -99999
    // Se genera el objeto fake a usar
    myFake := &fakeConfig{}
    // Se ejecuta el test y Se valida que para el cliente autorizado devuelva el resultado correcto de la suma y que para el cliente no autorizado devuelva el número -99999
    resultado := SumarRestricted(num1, num2, myFake, cliente)
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")
    resultado2 := SumarRestricted(num1, num2, myFake, cliente_dos)
    assert.Equal(t, resultadoEsperadoError, resultado2, "deben ser iguales")
}
```

Fake

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    go-testing/calc  1.085s
```

APLICANDO LO APRENDIDO

GO TESTING

Testearemos nuestro Service.go

Vamos a tomar el Proyecto Web y en el archivo service.go del folder internal/products, vamos a añadir una función Suma como se ve a continuación.

{}

```go
func (s *service) Sum(prices ...float64) float64 {
    var price float64
    for _, p := range prices {
        price += p
    }
    return price
}
```

Una función que recibe “n” cantidad de argumentos del tipo float64 y retorna la suma de todos los precios.

{}

```go
type DummyRepo struct{}

func (dr *DummyRepo) GetAll() ([]Product, error) {
    return []Product{}, nil
}
func (dr *DummyRepo) Store(id int, name, productType string, count int, price float64)
(Product, error) {
    return Product{}, nil
}
func (dr *DummyRepo) LastID() (int, error) {
    return 0, nil
}
func (dr *DummyRepo) UpdateName(id int, name string) (Product, error) {
    return Product{}, nil
}
func (dr *DummyRepo) Update(id int, name, productType string, count int, price float64)
(Product, error) {
    return Product{}, nil
}
func (dr *DummyRepo) Delete(id int) error {
    return nil
}
```

Testearemos nuestro Service.go

Para el test inicializamos el valor esperado. Creamos el objeto myDummyRepo, y este último lo usamos para crear el objeto myService. Se ejecuta el test con valores predefinidos y consistentes con el resultado esperado y se hacen las validaciones correspondientes.

{}

```go
func TestSum(t *testing.T) {
    expectedResult := float64(6)
    myDummyRepo := DummyRepo{}
    myService := NewService(&myDummyRepo)

    result := myService.Sum(1, 2, 3)

    assert.Equal(t, expectedResult, result)
}
```

$

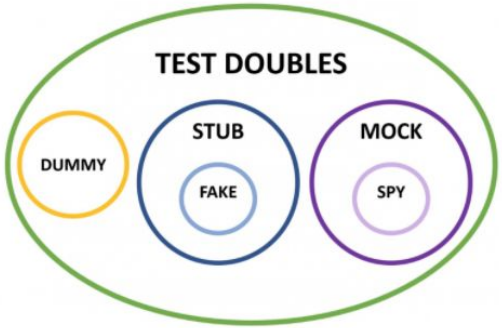
```
go test
```

output

```
PASS
ok      go-testing/calc     0.295s
```

Consideraciones Generales

Los Test Double se dividen en los distintos tipos previamente descritos, porque el objetivo de cada uno es diferente. Sin embargo, hay similitudes entre ellos que fundamentan la premisa que de cierta manera un Stub es ligeramente parecido a un Dummy, pero devuelve un valor específico. Un Spy es un tipo de Stub, pero con la responsabilidad adicional de guardar información. Un mock es una clase de Spy, pero en el que las validaciones se hacen sobre la información guardada en el Mock. Y un Fake es el rebelde que podría pasar por Stub pero se distingue porque contiene lógica de negocio, y devuelve distintas respuestas de acuerdo al escenario.



CUÁNDO APLICARLOS

GO TESTING

¿ Cuándo aplicar cada tipo ?

El objetivo de cada Test Double y el comportamiento es diferente. Naturalmente aplicar uno u otro depende exclusivamente de la necesidad del objeto de prueba (Función a testear). El testing en general procura comprobar la calidad del código, pero eso sin perder simplicidad y legibilidad en el código, por lo que la decisión siempre debe ser, usar el Double mas simple requerido para testear el objeto de prueba y todas sus condiciones y flujos. Evitar darle complejidad innecesaria a los tests y no descartar la continua posibilidad de refactorización o simplificación de los tests.

Gracias.

IT BOARDING

BOOTCAMP

Autor: Nelber Mora
Email: nelber.mora@digitalhouse.com
Última fecha de actualización: 08-07-21

IT BOARDING
BOOTCAMP