INTRODUCCIÓN A LA CLASE

GO TESTING

Objetivos de esta clase

- Introducirnos al concepto de Test de Dobles.
- Comprender la motivación de este tipo de test.
- Conocer los tipos de Test de Dobles.
- Implementar Test Doubles con Go.

TEST DOBLE

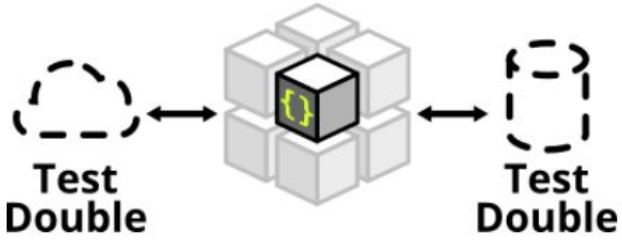
GO TESTING

// ¿Qué es un Test de Dobles?

Es un tipo de test unitario. Con su uso, se busca probar porciones de código que tengan dependencias de otros componentes, creando pequeñas piezas que suplan dicha dependencia.

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

Un objeto Dummy es algo que se utiliza para satisfacer dependencias, pero su uso en ejecución es completamente irrelevante.

Suelen implementarse como estructuras que implementen una interfaz específica, pero que su comportamiento sea completamente vacío. Los casos de uso de este tipo de test, son aquellos donde el objeto requerido no interviene en la ejecución de nuestro algoritmo.

Ejemplo de Dummy

Imaginemos el siguiente escenario:

- Debemos desarrollar un programa que realiza búsquedas en un directorio telefónico.
- Para poder instanciar nuestro servicio, debemos utilizar la función NewEngine, que recibe por parámetro una instancia del motor de búsqueda.
- Conocemos la interfaz del motor de búsqueda, y estamos implementando el servicio que se va a encargar de interactuar con este motor.
- Dentro de los requerimientos del servicio, se solicita debe implementarse un método GetVersion(), que devuelva un string con el nombre de la versión del servicio.

Implementación del ejemplo:

Para invocar ese método, debemos hacerlo de la siguiente forma:

{}

```
package main
import (
        ...
)

func main(){
   buscador := SearchEngine{}
   motor := NewEngine(buscador)

   fmt.Printf("La versión actual es %s", motor.GetVersion())
}
```

Testeando la funcionalidad (1/2)

Para poder realizar un test unitario que pruebe nuestro método, debemos proveerle un SearchEngine a la función NewEngine. Sin embargo, no queremos depender de que el SearchEngine esté listo y probado. Aquí va a entrar en juego un dummy del buscador, para poder suplir esa dependencia.

{}

```
type DummySearchEngine struct{}

func (d DummySearchEngine) BuscarPorNombre(nombre string) string {
   return ""
}

func (d DummySearchEngine) BuscarPorTelefono(telefono string) string {
   return ""
}

func (d DummySearchEngine) AgregarEntrada(nombre, telefono string) error {
   return nil
}
```

Testeando la funcionalidad (2/2)

Y ahora desarrollamos el test unitario utilizando el dummy en lugar de una instancia productiva del buscador

{}

```go
package directorio

import (
   "testing"

   "github.com/stretchr/testify/assert"
)

func TestGetVersion(t *testing.T) {
   //arrange

   myDummyDB := DummyDB{}
   motor := NewEngine(myDummyDB)
   versionEsperada := "1.0"

   //act
   resultado := motor.GetVersion()

   //assert
   assert.Equal(t, versionEsperada, resultado)

}
```

Dummy

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

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    directorio    0.516s
```

STUB TEST

GO TESTING

Stub

Un stub es un objeto similar a un dummy, con la diferencia de que este retornará valores concretos, para guiar la ejecución del código por un camino determinado

{}

```
type StubSearchEngine struct{}

func (d StubSearchEngine) BuscarPorNombre(nombre string) string {
   return "12345678"  // notar que en lugar de retornar "", retornamos un valor concreto
}

func (d StubSearchEngine) BuscarPorTelefono(telefono string) string {
   return ""
}

func (d StubSearchEngine) AgregarEntrada(nombre, telefono string) error {
   return nil
}
```

{}

```
func TestFindByName(t *testing.T) {
    //arrange

    myStubSearchEngine := StubSearchEngine{}
    motor := NewEngine(myStubSearchEngine)
    telefonoEsperado := "12345678"

    //act
    resultado := motor.FindByName("Pepe")

    //assert
    assert.Equal(t, telefonoEsperado, resultado)
}
```

Stub

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok    directorio    0.516s
```

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.

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.

En nuestro ejemplo, vamos a verificar si el método “BuscarPorTelefono” fue invocado

{}

```
type SpySearchEngine struct {
   BuscarPorTelefonoWasCalled bool
}

func (s *SpySearchEngine) BuscarPorNombre(nombre string) string {
   return ""
}

func (s *SpySearchEngine) BuscarPorTelefono(telefono string) string {
   s.BuscarPorTelefonoWasCalled = true
   return ""
}

func (s *SpySearchEngine) AgregarEntrada(nombre, telefono string) error {
   return nil
}
```

Realizamos el test con spy (1/2)

{}

```
func TestFindByTelephone(t *testing.T) {
   //arrange

   mySpySearchEngine := SpySearchEngine{BuscarPorTelefonoWasCalled: false}
   motor := NewEngine(&mySpySearchEngine)
   telefono := "12345678" // en nuestro motor, no realizaremos búsquedas si el telefono
                          // tiene una longitud menor a 5

   //act
   motor.FindByTelephone(telefono)

   //assert
   assert.True(t, mySpySearchEngine.BuscarPorTelefonoWasCalled)
}
```

Realizamos el test con spy (2/2)

{}

```
func TestFindByTelephoneNotCalled(t *testing.T) {
   //arrange

   mySpySearchEngine := SpySearchEngine{}
   motor := NewEngine(&mySpySearchEngine)

   //act
   motor.FindByTelephone("1234")   // el telefono tiene menos de 5 caracteres

   //assert
   assert.False(t, mySpySearchEngine.BuscarPorTelefonoWasCalled)
}
```

Spy

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output ok    directorio    0.174s
```

MOCK TEST

GO TESTING

Mock

El Mock, a diferencia de un Stub, no es aplicado únicamente para devolver valores exactos, sino para comprobar todo el funcionamiento interno del método o código que se está probando.

Un mock siempre es un espía y conoce lo que se está testeando. Y las comprobaciones del test se aplican sobre el mock.

Mock

Vamos a probar la funcionalidad de FindByName de nuestro motor, pero verificando que el método BuscarPorNombre haya sido invocado y haya devuelto un valor.

{}

```
type MockSearchEngine struct {
   BuscarPorNombreWasCalled bool
}

func (m *MockSearchEngine) BuscarPorNombre(nombre string) string {
   m.BuscarPorNombreWasCalled = true
   return "12345678"
}

func (m *MockSearchEngine) BuscarPorTelefono(telefono string) string {
   return ""
}

func (m *MockSearchEngine) AgregarEntrada(nombre, telefono string) error {
   return nil
}
```

Realizamos el test con mock (1/2)

{}

```
func TestFindByNameMocked(t *testing.T) {
   //arrange

   myMockSearchEngine := MockSearchEngine{}
   motor := NewEngine(&myMockSearchEngine)
   telefonoEsperado := "12345678"

   //act
   resultado := motor.FindByName("Nacho")

   //assert
   assert.Equal(t, telefonoEsperado, resultado)
   assert.True(t, myMockSearchEngine.BuscarPorNombreWasCalled)
}
```

Realizamos el test con mock (2/2)

{}

```
func TestFindByNameMockedNotCalled(t *testing.T) {
   //arrange

   myMockSearchEngine := MockSearchEngine{}
   motor := NewEngine(&myMockSearchEngine)
   telefonoEsperado := ""

   //act
   resultado := motor.FindByName("Nac")

   //assert
   assert.Equal(t, telefonoEsperado, resultado)
   assert.False(t, myMockSearchEngine.BuscarPorNombreWasCalled)
}
```

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      directorio    0.610s
```

FAKE TEST

GO TESTING

Fake

Los fakes son objetos que tienen una implementación funcional, pero no la misma que la que se utilizará en producción. Generalmente, suelen ser acercamientos simplificados al comportamiento real del componente.

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.

Implementando el Fake (1/2)

Tomando como objeto de pruebas nuestro directorio telefónico, podemos implementar un fake de nuestro SearchEngine, que realmente sea funcional, pero con features reducidas. En este caso, desarrollaremos una base de datos en memoria, para suplir un storage real.

{}

```
type FakeSearchEngine struct {
   DB map[string]string
}
```

Implementando el Fake (2/2)

{}

```go
func (m *FakeSearchEngine) BuscarPorNombre(nombre string) string {
    return m.DB[nombre]
}

func (m *FakeSearchEngine) BuscarPorTelefono(telefono string) string {
    for key, value := range m.DB {
        if value == telefono {
            return key
        }
    }
    return ""
}

func (m *FakeSearchEngine) AgregarEntrada(nombre, telefono string) error {
    if m.DB == nil {
        m.DB = map[string]string{}
    }
    m.DB[nombre] = telefono
    return nil
}
```

Realizando un test usando el Fake

```
{}
```

```go
func TestFindByNameFaked(t *testing.T) {
   //arrange

   myFakeSearchEngine := FakeSearchEngine{}
   motor := NewEngine(&myFakeSearchEngine)
   testValues := map[string]string{"Nacho": "123456", "Nico": "234567"}
   for key := range testValues {
       motor.AddEntry(key, testValues[key])
   }

   //act
   resultadoNacho := motor.FindByName("Nacho")
   resultadoNico := motor.FindByName("Nico")
   resultadoTelefono := motor.FindByTelephone("123456")

   //assert
   assert.Equal(t, testValues["Nacho"], resultadoNacho)
   assert.Equal(t, testValues["Nico"], resultadoNico)
   assert.Equal(t, "Nacho", resultadoTelefono)

}
```

Fake

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok      directorio   0.495s
```

Consideraciones Generales

Los Test Double se dividen en los distintos tipos previamente descritos, porque el objetivo de cada uno es diferente. Sin embargo, podemos notar similitudes entre ellos:

- *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 ?

Aplicar uno u otro depende exclusivamente de la necesidad y escenario del objeto de prueba.

El testing en general procura comprobar la calidad del código, pero eso sin perder simplicidad y legibilidad del mismo, por lo que la decisión siempre debe ser usar el Double más 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.

Conclusión

- Aprendimos cuál es propósito de los test dobles y qué son
- Conocimos los diferentes tipos de test doubles (dummy, stub, spy, mock y fake)
- Comprendimos el funcionamiento de los diferentes tipos de tests doubles
- Comprendimos cuando utilizarlos

Gracias.

IT BOARDING

BOOTCAMP

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

IT BOARDING
BOOTCAMP