¿Qué son los dobles de test?

Antonio Leiva
5 enero, 2023
11 min lectura

Comparte

Los dobles de prueba (también conocidos como "doubles" o "fakes") son herramientas comunes en la programación y en particular en el testing de software.

Se utilizan para simular el comportamiento de una dependencia de una aplicación en un entorno de pruebas, sin tener que depender de la implementación real de dicha dependencia.

Esto tiene varias ventajas. En primer lugar, permite realizar pruebas aisladas de una parte específica de una aplicación, sin tener que preocuparse por el comportamiento de sus dependencias.

Esto es especialmente útil cuando estamos trabajando con dependencias externas que pueden ser difíciles de controlar o replicar en un entorno de pruebas.

En segundo lugar, los dobles de prueba nos permiten controlar el comportamiento de las dependencias de nuestra aplicación.

Esto es útil para probar diferentes escenarios y ver cómo nuestra aplicación se comporta en cada uno de ellos.

Por ejemplo, podemos crear un double que simule una respuesta fallida de un servicio web y probar cómo nuestra aplicación maneja dicho fallo.

Tipos de dobles de test

Existen diferentes tipos de dobles de prueba, cada uno con sus propias características y usos.

Dummy

Los dobles de prueba "Dummy" son aquellos que son utilizados simplemente para proporcionar una implementación vacía para una dependencia necesaria en nuestro código.

Estos dobles son muy simples y no tienen ningún tipo de lógica en su implementación.

Se utilizan principalmente cuando necesitamos un objeto para cumplir con una firma de método o una interfaz, pero no tenemos intención de utilizar realmente ese objeto.

En pocas palabras, queremos hacer que compile.

Por ejemplo, imaginemos que tenemos una clase llamada `Customer` con un método llamado `makePayment()` que depende de un objeto `CreditCard` para realizar el pago.

En lugar de proporcionar una implementación real de `CreditCard`, podemos utilizar un Dummy para cumplir con la firma del método y poder probar nuestra clase `Customer` de forma aislada.

Aquí tienes un ejemplo de código en Kotlin que muestra cómo se podría utilizar un Dummy para proporcionar una dependencia a nuestra clase `Customer`:

```
class CreditCardDummy: CreditCard {
    // Implementación vacía de los métodos de CreditCa
```

```
    fun makePayment(creditCard: CreditCard) {
        // Lógica para realizar el pago con l
    }
}

// Utilizamos una instancia de CreditCardDummy como de
val customer = Customer()
val creditCard = CreditCardDummy()
customer.makePayment(creditCard)
```


Stub

Los dobles de prueba "Stub" son aquellos que se utilizan para simular el comportamiento de una dependencia en un entorno de pruebas. Estos dobles proporcionan respuestas predefinidas a ciertas entradas y son muy útiles para probar cómo nuestra aplicación se comporta en diferentes escenarios.

Por ejemplo, imaginemos que tenemos una clase `Customer` con un método `getDiscount()` que depende de un objeto `DiscountService` para obtener un descuento para un determinado cliente. En lugar de depender del servicio real, podemos utilizar un Stub para proporcionar respuestas predefinidas y así poder probar nuestra clase `Customer` de forma aislada.

Aquí tienes un ejemplo de código en Kotlin que muestra cómo se podría utilizar un Stub para proporcionar una dependencia a nuestra clase `Customer`:

```
class DiscountServiceStub: DiscountService {
    // Implementación del método getDiscount que siemp
    override fun getDiscount(customer: Customer): Doub
        return 0.1
    }
}

class Customer {
    fun getDiscount(discountService: DiscountService):
        // Lógica para obtener el descuento utilizando
        return discountService.getDiscount(this)
    }
}

// Utilizamos una instancia de DiscountServiceStub com
val customer = Customer()
```

Fake

Los dobles de prueba "Fake" son aquellos que imitan el comportamiento y la interfaz de una dependencia real, pero utilizan una implementación simplificada y más fácil de probar. Estos dobles son muy útiles cuando la dependencia es compleja o cuesta mucho trabajo configurarla para un entorno de pruebas.

Por ejemplo, imaginemos que tenemos una clase `Customer` con un método `sendEmail()` que depende de un objeto `EmailService` para enviar un correo electrónico a un cliente. En lugar de depender del servicio real, que podría ser complicado de configurar y utilizar en un entorno de pruebas, podemos utilizar un Fake que imite el comportamiento del servicio real pero que sea más fácil de probar.

Aquí tienes un ejemplo de código en Kotlin que muestra cómo se podría utilizar un Fake para proporcionar una dependencia a nuestra clase `Customer`:

```
class EmailServiceFake: EmailService {
    // Implementación del método sendEmail que simplem
    val sentEmails: MutableList<Email> = mutableListOf

    override fun sendEmail(email: Email) {
        sentEmails.add(email)
    }
}

class Customer {
    fun sendEmail(emailService: EmailService, email: E
        // Lógica para enviar el correo utilizando el
        emailService.sendEmail(email)
    }
}

// Utilizamos una instancia de EmailServiceFake como d
val customer = Customer()
val emailService = EmailServiceFake()
customer.sendEmail(emailService, Email(to = "john@exam
```

Para preparar un test, podríamos incluso añadir emails iniciales de forma manual y partir desde el punto que necesitemos

Spy

Estos dobles son muy útiles cuando queremos asegurarnos de que una dependencia se está utilizando correctamente y que está produciendo el resultado esperado.

Por ejemplo, imaginemos que tenemos una clase `Customer` con un método `addItemToCart()` que depende de un objeto `ShoppingCart` para añadir un producto al carrito de la compra.

En lugar de depender del carrito real, podemos utilizar un Spy para registrar el comportamiento del carrito durante la prueba y así poder verificar que se está utilizando correctamente.

Algunas definiciones afirman que se usa el objeto real, y simplemente añadimos código que nos permite verificar las llamadas.

Aquí tienes un ejemplo de código en Kotlin que muestra cómo se podría utilizar un Spy para proporcionar una dependencia a nuestra clase `Customer`:

```
class ShoppingCartSpy: ShoppingCart {
    // Implementación del método addItem que simplemen
    private val addedItems: MutableList<Product> = mut
    var addItemCalled = false

    override fun addItem(product: Product) {
        addItemCalled = true
        addedItems.add(product)
    }
}

class Customer {
    fun addItemToCart(shoppingCart: ShoppingCart, prod
        // Lógica para añadir el producto al carrito d
        shoppingCart.addItem(product)
    }
}

// Utilizamos una instancia de ShoppingCartSpy como de
val customer = Customer()
val shoppingCart = ShoppingCartSpy()
customer.addItemToCart(shoppingCart, Product(name = "P
```

La principal diferencia entre un fake y un spy es que un fake imita el comportamiento y la interfaz de una dependencia real, pero utiliza una implementación simplificada y más fácil de probar, mientras que un spy registra el comportamiento de una dependencia

de configurar o porque queremos probar nuestro código de forma aislada.

Por otro lado, un spy se utiliza para verificar que una dependencia se está utilizando correctamente y que está produciendo el resultado esperado.

Mock

Los dobles de prueba "Mock" son un tipo de doble de prueba que se utiliza para verificar el comportamiento de una dependencia durante una prueba. A diferencia de los fake y los spy, los mock tienen un comportamiento totalmente controlado por la prueba y no se basan en la implementación real de la dependencia.

Los mock se utilizan principalmente para asegurar que nuestro código se está comportando de la forma esperada cuando se produce un evento o se invoca una función en una dependencia. Al poder controlar el comportamiento del mock, podemos simular diferentes escenarios y verificar que nuestro código se está ejecutando correctamente en cada uno de ellos.

Los mocks por tanto son una evolución de los spies donde podemos controlar cada interacción y proveer el código que queremos que se ejecute ante cada llamada.

Normalmente se utilizan librerías que los implementan por nosotros. Las más populares en Kotlin son Mockito y MockK.

```
class ShoppingCart(private val priceCalculator: PriceC
    fun checkout(items: List<Item>) =
        priceCalculator.calculateTotalPrice(items)
    }
}

// Creamos un Mock de PriceCalculator utilizando Mocki
val priceCalculatorMock = mock<PriceCalculator> {
    on { calculateTotalPrice(any()) } doReturn 100.0
}

// Utilizamos el Mock como dependencia
val shoppingCart = ShoppingCart(priceCalculatorMock)
```

Ejemplo real de creación de los distintos dobles de test

Cuando quiere recuperar las películas a mostrar, buscará primero en local.

- Si encuentra contenido lo devolverá.
- Si no, irá al servidor, lo descargará y lo guardará en local, devolviendo entonces el contenido local

```
data class Movie(val id: Int, val title: String)

class MoviesRepository(
    private val moviesLocaDataSource: MoviesLocalDataS
    private val moviesRemoteDataSource: MoviesRemoteDa
) {

    fun findAll(): List<Movie> {
        if (moviesLocaDataSource.isEmpty()) {
            val movies = moviesRemoteDataSource.findPo
            moviesLocaDataSource.saveAll(movies)
        }
        return moviesLocaDataSource.findAll()
    }

}

interface MoviesLocalDataSource {
    fun isEmpty(): Boolean
    fun saveAll(movies: List<Movie>)
    fun findAll(): List<Movie>
}

interface MoviesRemoteDataSource {
    fun findPopularMovies(): List<Movie>
}
```

Imagina que quieres testear el primer caso:

```
class MoviesRepositoryTest {

    @Test
```

En este caso, el DataSource remoto no me interesa, ya que solo voy a usar el local. Por tanto, puede ser un *dummy*:

```
class MoviesRemoteDataSourceDummy : MoviesRemoteDataSo
    override fun findPopularMovies(): List<Movie> = TO
}
```

Sin embargo, del local necesito que devuelva algo, y por tanto, tiene que ser al menos un *Stub* que me devuelva siempre el mismo valor (aquí no voy a probar a guardar):

```
class MoviesLocalDataSourceStub : MoviesLocalDataSourc
    override fun isEmpty(): Boolean = false
    override fun findAll(): List<Movie> = listOf(Movie
    override fun saveAll(movies: List<Movie>) = TODO()

}
```

El último método ni me molesto en implementarlo, porque no lo voy a usar. El test podría ser:

```
    @Test
    fun `getMovies() returns a list of local movies if
        val moviesLocalDataSource = MoviesLocalDataSou
        val moviesRemoteDataSource = MoviesRemoteDataS
        val moviesRepository = MoviesRepository(movies

        val movies = moviesRepository.findAll()

        Assert.assertEquals(1, movies[0].id)
    }
```

Pero imagina que quiero probar que si la fuente local está vacía, almacena lo que está en la remota. Ahora la fuente remota necesita ser un *Stub*:

```
class MoviesRemoteDataSourceStub : MoviesRemoteDataSou
```

```
)
```


Pero ahora la fuente local no puede ser tan simple, porque necesita poder guardar valores y devolverlos para que el código funcione y haga su trabajo.

Es decir, necesito una versión sencilla de la implementación real, que puede ser simplemente una persistencia en memoria.

A esto se lo conoce como *Fake*:

```
class MoviesLocalDataSourceFake : MoviesLocalDataSourc

    private val movies = mutableListOf<Movie>()

    override fun isEmpty(): Boolean = movies.isEmpty()

    override fun saveAll(movies: List<Movie>) {
        this.movies.addAll(movies)
    }

    override fun findAll(): List<Movie> = movies

}
```

Finalmente, el test sería:

```
    @Test
    fun `getMovies() returns a list of remote movies i
        val moviesLocalDataSource = MoviesLocalDataSou
        val moviesRemoteDataSource = MoviesRemoteDataS
        val moviesRepository = MoviesRepository(movies

        val movies = moviesRepository.findAll()

        Assert.assertEquals(1, movies[0].id)
        Assert.assertEquals(2, movies[1].id)
    }
```

Esto está muy bien, ¿pero cómo me aseguro de que lo que me devuelve el repositorio es realmente lo que está en el datasource local, y que se está llamando a las funciones que

información sobre si se ha llamado a la función que me interesa:


```
class MoviesLocalDataSourceSpy : MoviesLocalDataSource

    val movies = mutableListOf<Movie>()
    var saveAllCalled = false

    override fun saveAll(movies: List<Movie>) {
        saveAllCalled = true
        this.movies.addAll(movies)
    }

    ...

}
```

Ahora en el test podemos comprobar eso:

```
Assert.assertEquals(2, moviesLocalDataSource.movies.si
Assert.assertTrue(moviesLocalDataSource.saveAllCalled)
```

Pero este se puede volver un poco tedioso cuando tenemos muchas funciones y distintas funciones que espiar.

Así que podemos dar el salto y usar una librería como Mockito para crear el objeto. En el primer test, podríamos hacer:

```
val moviesLocalDataSource = mock<MoviesLocalDataSource
    on { isEmpty() } doReturn false
    on { findAll() } doReturn listOf(Movie(1, "title1"
}
```

Y finalmente comprobar que se ha llamado a la función que queremos:

```
verify(moviesLocalDataSource).findAll()
```

El caso del segundo test sería más complicado hacerlo con mocks, ya que el estado de un componente cambia a partir de la interacción con otro, y por tanto ya estamos testeando cómo se integran varios componentes.

En lo personal, normalmente distingo entre *Mocks* y *Fakes* y nada más, pero está bien que conozcas las sutilezas de cada uno de ellos para saber hasta qué detalle entrar en cada caso.

Conclusión

En conclusión, los dobles de test son una herramienta muy útil en el desarrollo de aplicaciones con Kotlin.

Existen varios tipos de dobles de test, cada uno con sus propias ventajas y usos específicos.

En el ejemplo del repositorio de películas, se ha visto cómo se pueden utilizar diferentes tipos de dobles de test en diferentes etapas del desarrollo, lo que permite probar el código de manera más eficiente y aislar problemas específicos.

Es importante tener en cuenta que el uso adecuado de los dobles de test puede mejorar significativamente la calidad del código y hacer que el proceso de desarrollo sea más eficiente.

Recursos de expertos para la solución de problemas

Ver todos

Principios SOLID: Qué son, cuáles, y qué beneficios aporta usarlos

Cápsulas Curso Arquitecturas Gratis

Patrones de diseño de software

Cápsulas Curso Arquitecturas Gratis

Cómo convertirse en un desarrollador de software: habilidades y conocimientos necesarios

Desarrollo Profesional Tips

Únete a nuestra comunidad y acelera tu crecimiento profesional con la ayuda de expertos.

Email

Unirme

Acepto la Política de Privacidad*

DevExpert

Guía gratuita

Cursos
Gratis
Expert

Recursos
Guías
Cápsulas
Tips

Comunidad
Eventos
Desarrolladores
Historias
DevsLetter

Nosotros
DevExpert
Antonio Leiva
Patrocinio
Contacto

Empresas
Formación

Legal
Condiciones de venta
Política de cookies
Política de privacidad
Términos de uso

Escríbenos

16:17:19