Dobles de pruebas

Definición | Entrada/Salida indirecta | Tipos | Dummy | Stub | Spy | Fake | Mock | EasyMock

Dobles de prueba

Cuando testeamos porciones de software incompleto, se necesitan componentes extras de software.

> Un doble es un esqueleto o una implementación especial de un módulo que necesito para testear un componente que depende de alguna manera de ese doble.

Dobles de prueba

Normalmente el funcionamiento del SUT (Software Under Test) depende de otros componentes, por dos vías:

- Entrada indirecta: es un valor obtenido por invocaciones a un método de un componente del cual se depende (Depended On Component DOC).
- Salida indirecta: es una potencial modificación al estado de un componente del cual de depende (DOC).

Dobles de prueba

Un doble de prueba reemplaza un DOC, aislado el SUT cuando:

- es necesario controlar las entradas indirectas, para manejar el hilo de ejecución que se desea ejercitar,
- es necesario monitorear las salidas indirectas, que son consecuencia del funcionamiento del SUT.

Ejemplo

Queremos probar la clase Order y OrderLine que dependen de la interfaz IShopDataAccess:

```
public class Order {

   private int id;
   private IShopDataAccess dataAccess;
   private List<OrderLine> orderLines;

   public OrderLineCollection getLines() {
   return orderLines;
  }

  public IShopDataAccess getDataAccess() {
   return dataAccess;
  }

   public void save() {
       this.dataAccess.save(this.id, this);
   }

   public Order(int id, IShopDataAccess dataAccess) {
       if (dataAccess == null)
         throw new ArgumentNullException("dataAccess");

       this.id = id;
       this.dataAccess = dataAccess;
       this.orderLines = new OrderLineCollection(this);
   }
   ...
```

```
public class OrderLine {
    private int id;
    private int quantity;
    private Order owner;

    public OrderLine(Order owner) {
        if (owner == null)
            throw new IllegalArgumentException(“owner");
        this.owner = owner;
    }

    public double getTotal() {
        double unitPrice =
            owner.getDataAccess().getProductPrice(id);
        double total = unitPrice * quantity;
        return total;
    }
    ...
```

```
public interface IShopDataAccess {
    double getProductPrice(int productId);
    void save(int orderId, Order o);
}
```

Tipos de dobles

Tipo
Dummy
Stub
Spy
Fake
Mock

Dummy

Un dummy sólo satisface las dependencias formales.

```
public class DummyShopDataAccess implements IShopDataAccess {

   public double getProductPrice(int productId) {
    throw new Exception("The method or operation is not implemented.");
   }

   public void save(int orderId, Order o) {
    throw new Exception("The method or operation is not implemented.");
   }

}
```

```
@Test
public void createOrder() {
    DummyShopDataAccess dataAccess = new DummyShopDataAccess();

    Order o = new Order(2, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    assertEquals(2, o.getLines().size());
}
```

El dummy es suficiente porque la interfaz nunca es ejercitada.

Stub

Si el test invoca algún método es necesario (al menos) **no** levantar una excepción.

```
public class StubShopDataAccess implements IShopDataAccess {

  public double getProductPrice(int productId) {
   throw new Exception("The method or operation is not implemented.");
  }

  public void save(int orderId, Order o) {
  }

}
```

```
@Test
public void saveOrder() {
    StubShopDataAccess dataAccess = new StubShopDataAccess();

    Order o = new Order(3, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    o.save();
}
```

```
public class Order {

   ...

   public void save() {
        this.dataAccess.save(this.id, this);
    }

   ...
```

Stub

La diferencia es más marcada cuando se invoca un método que devuelve un valor.

La implementación más sencilla es devolver valores fijos.

Controlando el entrada indirecta es posible verificar el comportamiento esperado.

```
public class OrderLine {
    private int id;
    private int quantity;
    private Order owner;

    public OrderLine(Order owner) {
        if (owner == null)
            throw new ArgumentNullException("owner");
        this.owner = owner;
    }

    public double getTotal() {
        double unitPrice =
            owner.getDataAccess().getProductPrice(id);
        double total = unitPrice * quantity;
        return total;
    }
    ...
```

```
public class StubShopDataAccess implements IShopDataAccess {

    public double getProductPrice(int productId) {
            return 25;
    }

    public void save(int orderId, Order o) { }
}
```

```
@Test
public void calculateSingleLineTotal() {
    StubShopDataAccess dataAccess = new StubShopDataAccess();

    Order o = new Order(4, dataAccess);
    o.getLines().add(1234, 2);

    double lineTotal = o.getLines().get(0).getTotal();
    assertEquals(50, lineTotal, 0.01);
}
```

Stub

La diferencia es más marcada cuando se invoca un método que devuelve un valor.

La implementación más sencilla es devolver valores fijos.

Controlando el entrada indirecta es posible verificar el comportamiento esperado.

```
public class OrderLine {
    private int id;
    private int quantity;
    private Order owner;

    public OrderLine(Order owner) {
        if (owner == null)
            throw new ArgumentNullException("owner");
        this.owner = owner;
    }

    public double getTotal() {
        double unitPrice =
            owner.getDataAccess().getProductPrice(id);
        double total = unitPrice * quantity;
        return total;
    }
    ...
```

¿Cómo flexibilizar el entrada indirecta?

¿Cómo verificar el salida indirecta?

```
public class StubShopDataAccess implements IShopDataAccess {

    public double getProductPrice(int productId) {
            return 25;
    }

    public void save(int orderId, Order o) { }
}
```

```
@Test
public void calculateSingleLineTotal() {
    StubShopDataAccess dataAccess = new StubShopDataAccess();

    Order o = new Order(4, dataAccess);
    o.getLines().add(1234, 2);

    double lineTotal = o.getLines().get(0).getTotal();
    assertEquals(50, lineTotal, 0.01);
}
```

Salida Indirecta: Spy

Verificar la salida indirecta requiere registrar las invocaciones y sus parámetros.

```
public class SpyShopDataAccess implements IShopDataAccess {
   private boolean saveInvoked;

   public void save(int orderId, Order o) {
      saveInvoked = true;
   }

   public boolean getSaveWasInvoked() {
          return this.saveInvoked;
   }

}
```

```
@Test
public void saveOrderWithDataAccessVerification() {
    SpyShopDataAccess dataAccess = new SpyShopDataAccess();

    Order o = new Order(5, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);
    o.save();

    assertTrue(dataAccess.getSaveWasInvoked());
}
```

Input indirecto: Fake

Flexibilizar el entrada indirecta implica aproximarse a una implementación de producción.

```
public class FakeShopDataAccess implements IShopDataAccess {
    private ProductCollection products;

    public FakeShopDataAccess() {
        this.products = new ProductCollection();
    }

    public double getProductPrice(int productId) {
        if (this.products.contains(productId)) {
            return this.products.get(productId).getUnitPrice();
        }
        throw new ArgumentOutOfRangeException("productId");
    }

    List<Product> getProducts() {
       return this.products;
    }

    public void save() {
       ...
    }
    ...
```

```
@Test
public void calculateLineTotalsUsingFake() {
    FakeShopDataAccess dataAccess = new FakeShopDataAccess();
    dataAccess.getProducts().add(new Product(1234, 45));
    dataAccess.getProducts().add(new Product(2345, 15));

    Order o = new Order(6, dataAccess);
    o.getLines().add(1234, 3);
    o.getLines().add(2345, 2);

    assertEquals(135, o.getLines().get(0).getTotal(), 0.01);
    assertEquals(30, o.getLines().get(1).getTotal(), 0.01);
}
```

Mocks

Denominación general para dobles que controlan entrada y salida indirecta.

Escribir mocks manualmente es una tarea ardua y propensa a errores.

En general los mocks se crean en tiempo de ejecución con la ayuda de una librería específica, que permite:

- Especifica el comportamiento esperado.
- Crea el objeto cuyos métodos serán invocados.
- Verifica el comportamiento ejercitado respecto al especificado.

De esta manera, no es necesario escribir el código que implementa el mock.

Tipos de dobles

Tipo	Descripción
Dummy	El más simple y primitivo. Son derivaciones de interfaces que no contienen ninguna implementación. Se utilizan normalmente como valores para parámetros que nunca se utilizan.
Stub	Contiene implementaciones mínimas, normalmente devolviendo valores constantes.
Spy	Almacena información sobre los métodos invocados para realizar verificaciones sobre ellos.
Fake	Contiene una implementación más compleja, manejando múltiples interacciones, asemejándose a una implementación de producción.
Mock	Es una implementación dinámica que puede configurarse para tener un comportamiento específico.

Algunas Librerías para Mocks

Algunas Librerías para Mocks

- Mockito

Algunas Librerías para Mocks

- Mockito
- JMock

Algunas Librerías para Mocks

Mockito

JMock

EasyMock

Algunas Librerías para Mocks

Mockito

JMock

EasyMock

EasyMock

Los ejemplos anteriores (Spy y Fake) se implementan fácilmente con EasyMock:

```
@Test
public void saveOrderAndVerifyExpectations() {
    IShopDataAccess dataAccess = createMock(IShopDataAccess.class);

    Order o = new Order(6, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    // Record expectations
    dataAccess.save(6, o);
    replay(dataAccess);

    o.save();

    verify(dataAccess);
}
```

EasyMock

Los ejemplos anteriores (Spy y Fake) se implementan fácilmente con EasyMock:

EasyMock

Los ejemplos anteriores (Spy y Fake) se implementan fácilmente con EasyMock:

```
@Test
public void calculateLineTotalsUsingMock() {
    IShopDataAccess dataAccess = createMock(IShopDataAccess.class);

 //Record expectations
 expect(dataAccess.getProductPrice(1234)).andReturn(45.0);
 expect(dataAccess.getProductPrice(2345)).andReturn(15.0);


 replay(dataAccess);

 Order o = new Order(11, dataAccess);
 o.getLines().add(1234, 3);
 o.getLines().add(2345, 2);

 assertEquals(135, o.getLines().get(0).getTotal(), 0.01);
 assertEquals(30, o.getLines().get(1).getTotal(), 0.01);


 //Make sure everything that was supposed to be called was called
 verify(dataAccess);
}
```

Funciones principales de EasyMock

createMock(<class>): crea un mock que implementa la interfaz <class>, sin registra del orden de invocación.

createNiceMock(<class>): crea un mock que implementa la interfaz <class>, sin orden de invocación y que devuelve 0, null o false para las invocaciones no esperadas.

createStrictMock(<class>): crea un mock que implementa la interfaz <class>, con registro del orden de invocación.

expect(<inv>): registra la expectativa de llamada a <inv>.

expect(<inv>).andReturn(<val>): además establece como resultado <val>.

expect(<inv>).andThrow(<exc>): además establece la ocurrencia de la excepción <exc>.

Funciones principales de EasyMock

expectLastCall.times(<n>): establece la expectativa de <n> llamadas a la ultima invocación registrada.

replay(<mock>): establece el comportamiento especificado sobre el <mock>.

verify(<mock>): verifica que se ejercite el comportamiento especificado <mock>.

reset(<mock>): resetea el comportamiento especificado sobre <mock>, útil para fixtures compartidos.

Bibliografia Recomendada

The art of unit testing, Roy Osherove, Cáp. 3 y 4.

The Art of Software Testing, Myers, Cáp. 5.

http://easymock.org/user-guide.html