El testing como parte del proceso de calidad del software

Dobles de testing

Contenido

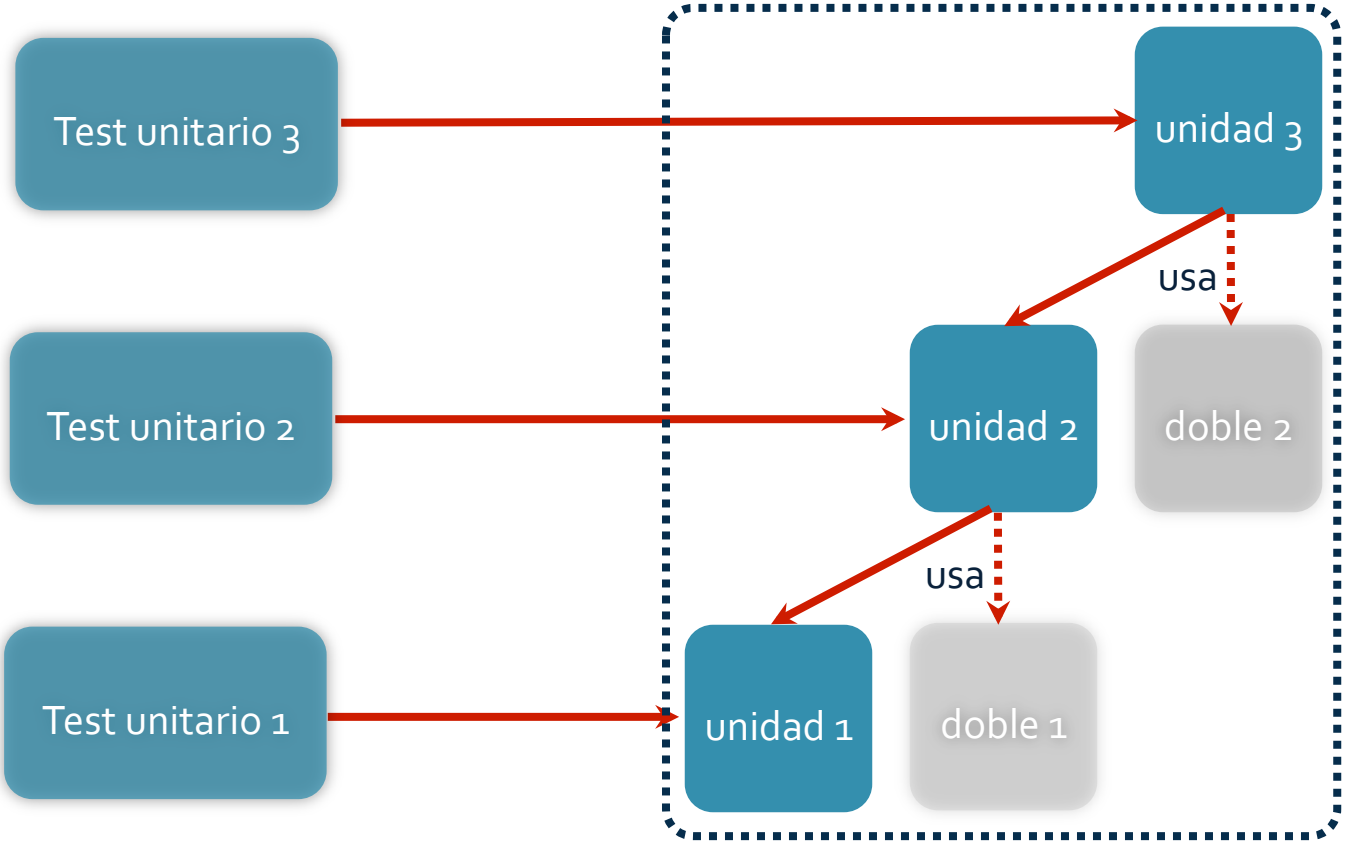
- ?

Dobles de testing

- Normalmente el funcionamiento del SUT depende de otros componentes, por dos vias:
  - Entrada indirecta: es un valor obtenido por invocaciones a un método de un DOC.
  - Salida indirecta: es una potencial modificación al estado de un DOC.
- Un doble de prueba reemplaza un DOC 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.

Estrategias de integración

Tradicionalmente un proyecto de software se integra de adentro hacia afuera:

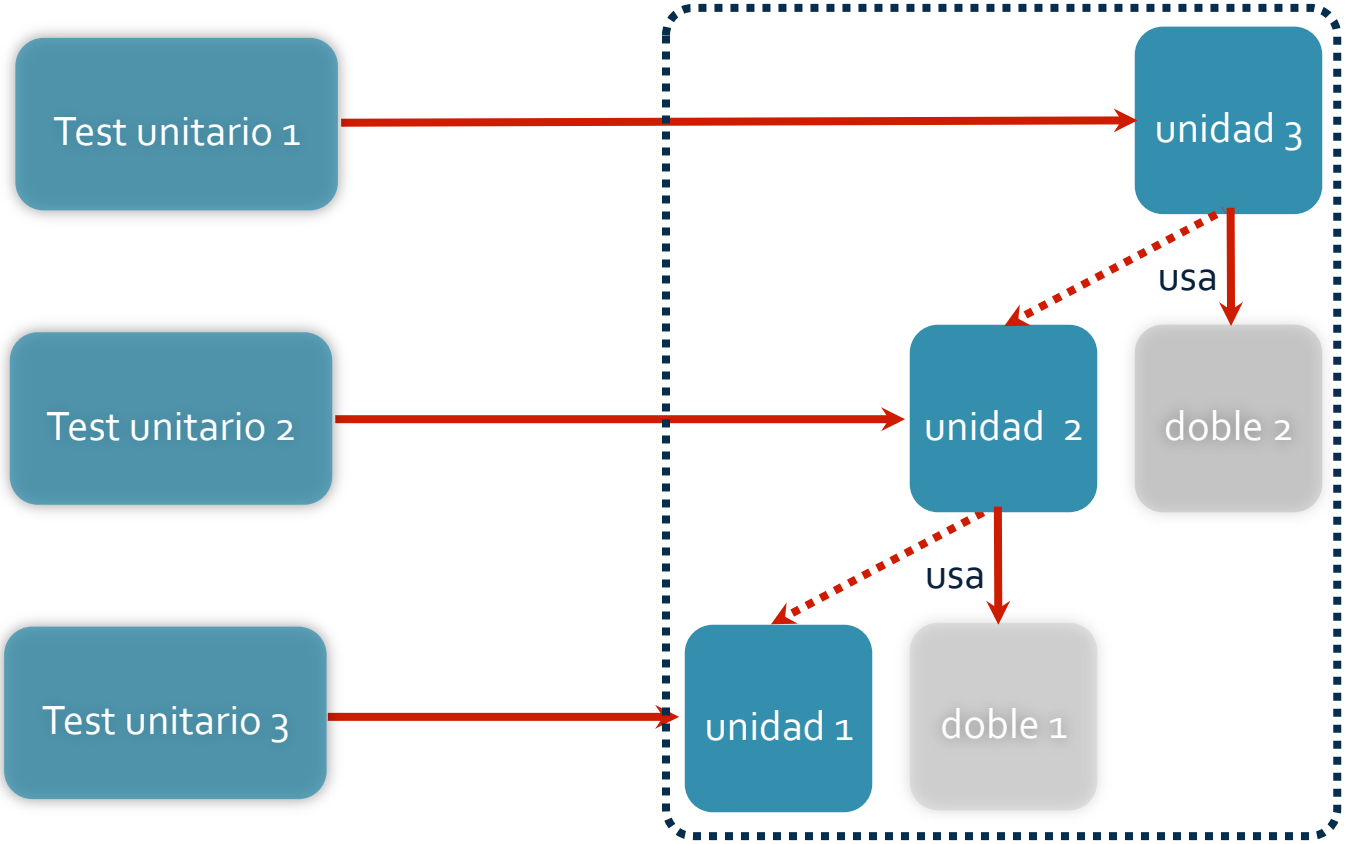


Adentro hacia afuera

- La codificación (y el testing) comienza en los componentes mas internos y simples, y prosigue hacia los mas complejos.
- Requiere un diseño y arquitectura mas detallado, y requerimientos bien especificados.
- Responde a un proceso de desarrollo tradicional.
- Minimiza la necesidad de dobles, pero minimiza el feedback temprano.
- Cambios imprevistos en los requerimientos por la falta de feedback repercuten en el re-trabajo sobre el testing.

Estrategias de integración

El testing unitario se acomoda bien con la estrategia de afuera hacia adentro:



Afuera hacia adentro

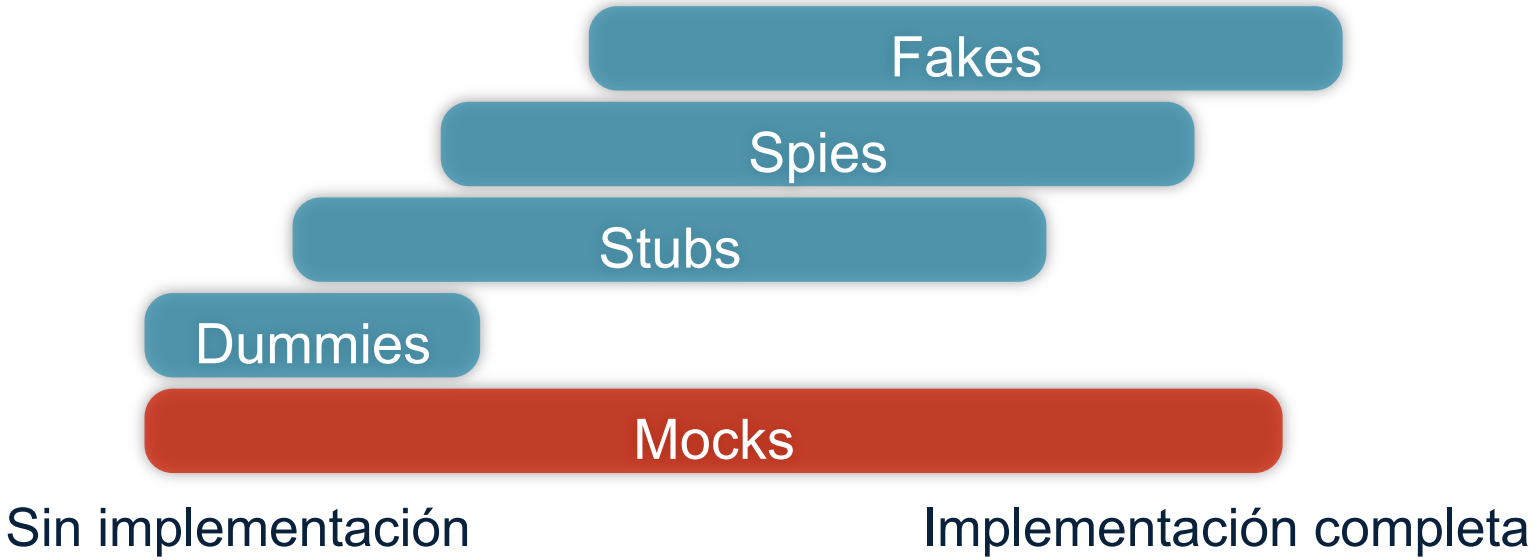
- La codificación (y testing) comienza en los componentes mas generales (prototipado).
- Los dobles se van reemplazando a medida que se avanza en la codificacion de los componentes.
- La arquitectura y diseño se van (re)definiendo "en la marcha".
- Responde a un proceso de desarrollo ágil.
- Maximiza la necesidad de dobles y el feedback, pero minimiza el re-trabajo por cambios en los requerimientos.

Tipos de dobles

Tipo	Descripción
Dummy	El mas simple y primitivo. Interfaces sin implementación. Para dependencias formales.
Stub	Implementaciones mínimas, que devuelven valores constantes.
Spy	Almacena información sobre los métodos invocados.
Fake	Implementación mas compleja, manejando múltiples interacciones.
Mock	Implementación dinámica que puede configurarse para un comportamiento específico.

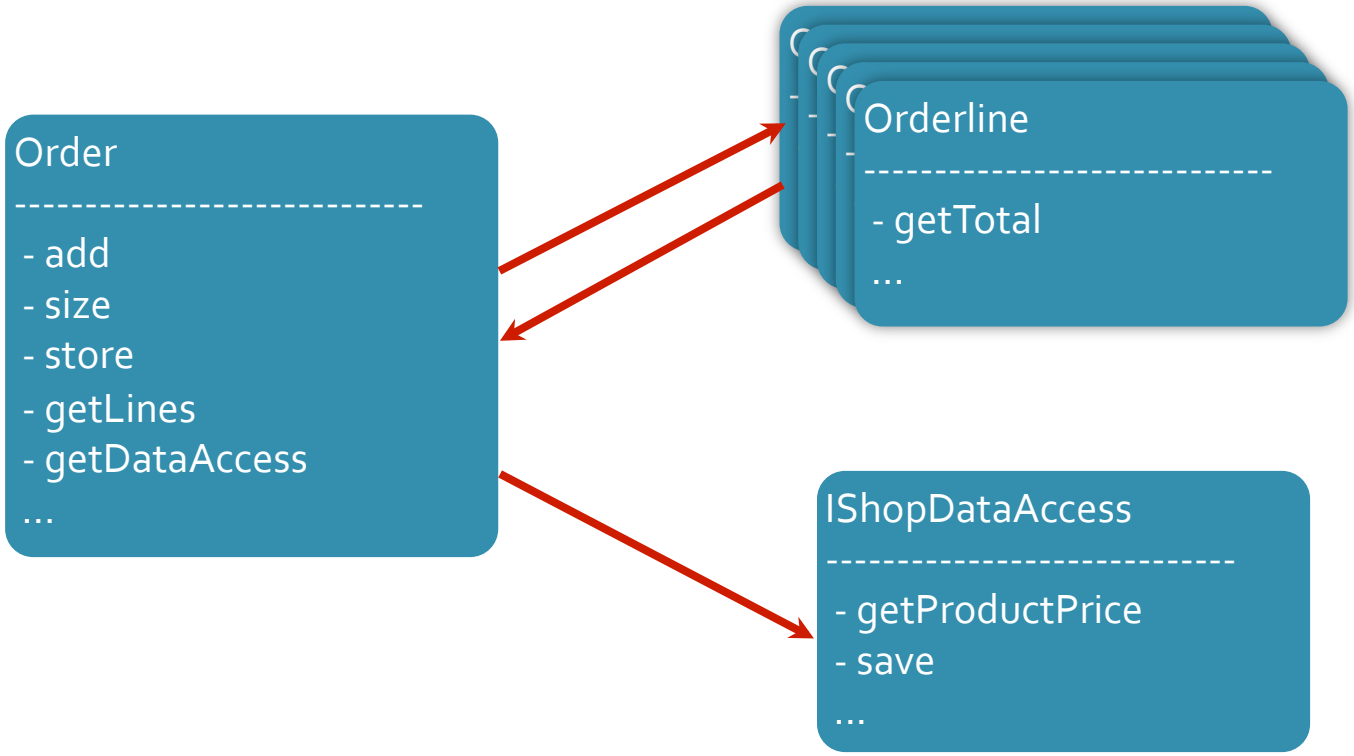
Tipos de dobles

- Aunque los tipos parecen diferentes en teoría, en la práctica las diferencias se vuelven borrosas.
- Parece mas apropiado, pensar los dobles como miembros de un contínuo:



Dobles: un ejemplo

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



Dobles: un ejemplo

```
public class Order {

    private int id;
    private IShopDataAccess dataAccess;
    private Collection<OrderLine> order

    public Collection<Orderline> getLir
         return orderLines;
    }

    public IShopDataAccess getDataAcces
         return dataAccess;
    }

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

    public Order(int id, IShopDataAcces
        if (dataAccess == null)
          throw new ArgumentNullExcepti

        this.id = id;
        this.dataAccess = dataAccess;
        this.orderLines = new Collection<Orde
    }
...
```

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

Dummy

- Un dummy sólo satisface las dependencias formales.

```
public class DummyShopDataAccess implements IShopDataAccess {

    public double ge
        throw new Ex
    }

    public void save
        throw new Ex
    }

}
```

```
@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());
}
```

- 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 get
        throw new Exce
    }

    public void save(
    }

}
```

```
@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.store();
}
```

- ¿Es esto un dummy o un stub?

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 input indirecto es posible verificar el comportamiento esperado.

Stub

```
public class OrderLine {
    private int id;
    private int quantity;
    private Order owner;
    public OrderLine(Order
        if (owner == null)
            throw new Argu
        this.owner = owner;
    }

    public d
        doub

        doub
        retu
    }
    ...
```

¿Cómo flexibilizar el input indirecto?

```
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);
}
```

¿Cómo verificar el output indirecto?

Spy

- Verificar el output indirecto requiere registrar las invocaciones y sus parámetros.

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

    public vo
        saveWa
    }

    public bo
        return

}
```

```
@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.store();

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

Fake

- Flexibilizar el input indirecto implica aproximarse a una implementación de producción.

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

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

    public double getPro
        if (this.product
            return this.
        }
        throw new Argume
    }

    List<Product> getPro
        return this.prod
    }

    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

- Mock es una denominación general para dobles que controlan entrada y salida indirecta.
- En general los mocks se crean en tiempo de ejecución con la ayuda de un framework:
  - Se crea el objeto cuyos métodos serán invocados.
  - Se especifica el comportamiento esperado.
  - Se verifica el comportamiento ejercitado respecto al especificado.
- No mezclar las fases!
- Así, no es necesario escribir el código que implementa el mock.

EasyMock

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

```
@Test
public void save
    IShopDataAcc

    Order o = ne
    o.getLines()
    o.getLines()

    // Record ex
    dataAccess.s
    replay(dataA

    o.store();

    verify(dataA
}
```

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

   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);

}
```

EasyMock

- createMock(<class>): crea un mock que implementa la interfaz <class>, sin registro del orden de invocación.
- createNiceMock(<class>): crea un mock que implementa la interfaz <class>, sin orden de invocación y que devuelve o, 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>.

EasyMock

- expectLastCall.times(<n>): establece la expectativa de <n> llamadas a la ultima invocación registrada.
- anyInt( ), anyChar( ), anyObject( )...: reemplazan los parámetros de las invocaciones que se registran.
- replay(<mock>): establece el comportamiento especificado sobre el <mock>.
- verify(<mock>): verifica el comportamiento especificado sobre <mock>.
- reset(<mock>): resetea el comportamiento especificado sobre <mock>, útil para fixtures compartidos.

Ejercicio

La clase IPBlackList posee un método *login* que se comporta según las siguientes reglas:

- almacena el ultimo IP que intento *loguearse* usando LoginService.
- almacena el número de intentos fallidos consecutivos del mismo IP.
- si un mismo IP falla tres veces consecutivas al intentar loguearse, esta IP se almacena en una lista negra.

1. Generá un caso de prueba en el que el *login* se realice correctamente al segundo intento.
2. Generá un caso de prueba de manera tal que la IP con la que intenta *loguearse* figure en la lista negra.

Introducción al testing unitario