Testing de unidad

Dra. Alejandra Garrido
Objetos 2 – Fac. De Informática – U.N.L.P.
alejandra.garrido@lifia.info.unlp.edu.ar

Testeo todos los aspectos / valores en 1 método

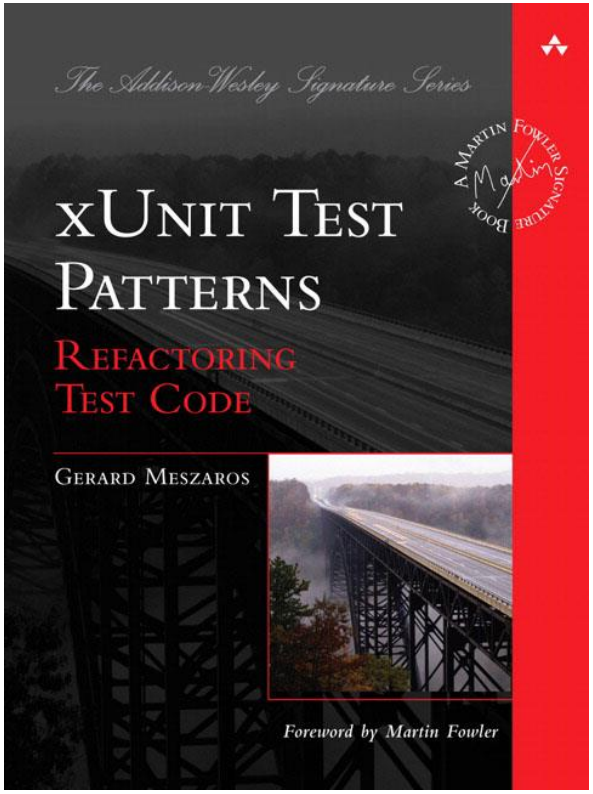
testFactorial

```
self assert: (small factorial = 2).
self assert: (big factorial =  3628800).
self should: [neg factorial] raise: Error
self assert: (zero factorial = 1)
```

O testeo 1 aspecto por vez

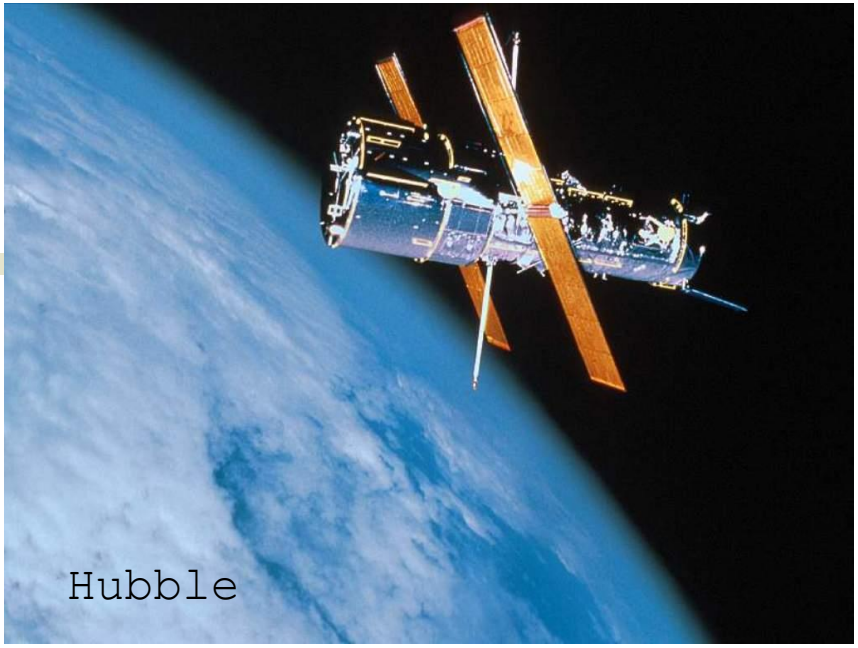
```
testSmallFactorial
   self assert: (small factorial = 2)

testBigFactorial
   self assert: (big factorial =  3628800)

testNegNumberFactorial
   self should: [neg factorial] raise: Error

testZeroFactorial
    self assert: (zero factorial = 1)
```

XUnit Test Patterns



Tamaño de los métodos de testing

- Postura más purista: verificar una sola condición por cada test.
- Ventaja para detectar errores: cuando un test falla se puede saber con precisión qué está mal con el SUT.
- Un test que verifica una única condición ejecuta un solo camino en el código del SUT (cobertura)
- Desventaja: el costo de testear cada camino de ejecución es demasiado alto! Cobertura del 100% es inviable
- Vale la pena?



Ariane 5 con componentes de Ariane 4
(error de overflow sin handler)

Tamaño de las clases de testing

- Una subclase de TestCase por cada clase
  - *TestInteger*
- Una clase testcase por cada método (feature)
  - *TestIntegerFactorial*
- Una clase testcase por cada fixture
  - *TestLargeIntegerFactorial*

Qué se hace en Fixture setup?

- La lógica del *fixture setup* incluye:
  - El código para instanciar el SUT
  - El código para poner el SUT en el estado apropiado
  - El código para crear e inicializar todo aquello de lo que el SUT depende o que le va a ser pasado como argumento

FlighStateTestCase>>testStatusInitial

“in-line setup”

departureAirport := Airport newIn: ‘Calgary’ name: ‘YYC’.

destinationAirport := Airport newIn: ‘Toronto’ name: ‘YYZ’.

flight := Flight newNumber: ‘0572’
from: departureAirport to: destinationAirport.

“exercise SUT and verify outcome”

self assert: (flight getStatus = ‘PROPOSED’)

}

FlighStateTestCase>>testStatusCancelled

“in-line setup”

departureAirport := Airport newIn: ‘Calgary’ name: ‘YYC’.

destinationAirport := Airport newIn: ‘Toronto’ name: ‘YYZ’.

flight := Flight newNumber: ‘0572’
from: departureAirport to: destinationAirport.

flight cancel.

“exercise SUT and verify outcome”

self assert: (flight getStatus = ‘CANCELLED’).

}

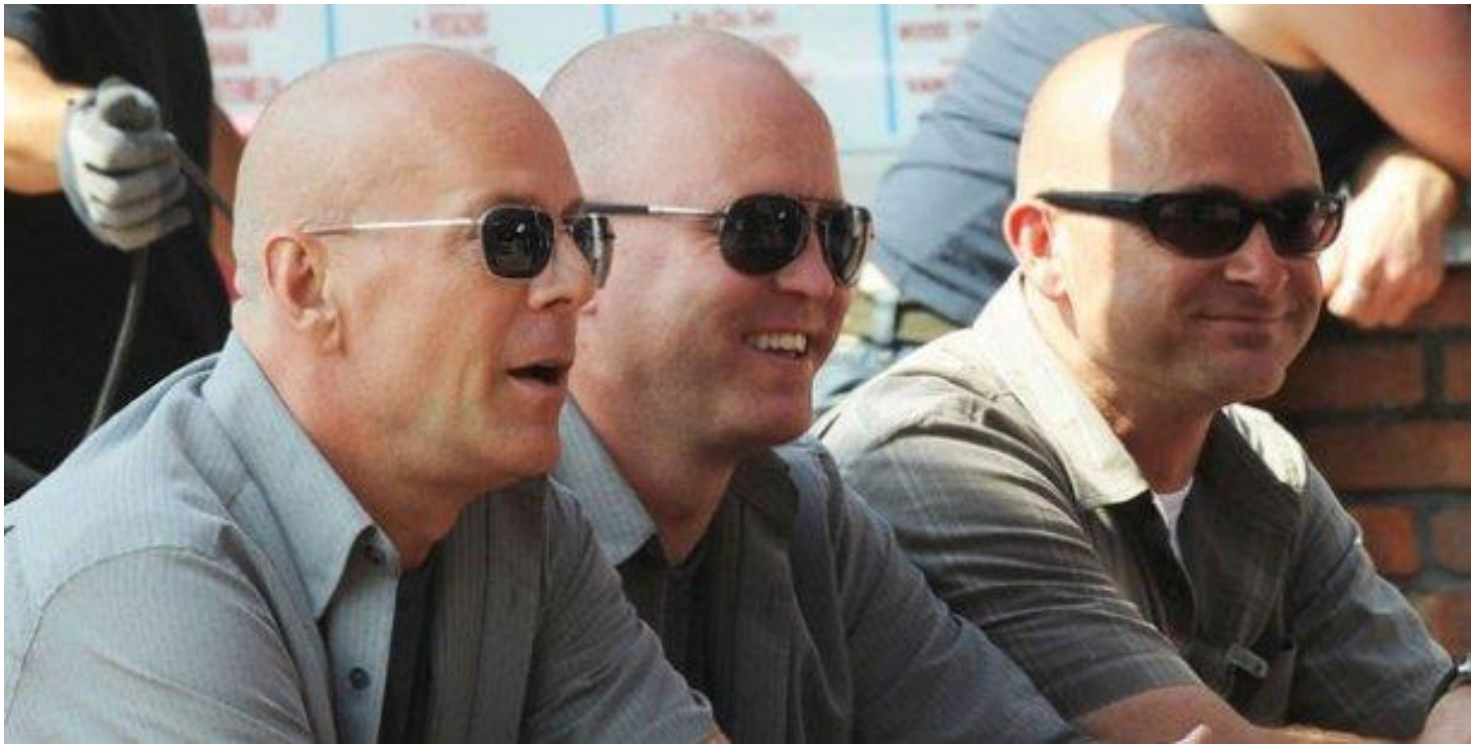
“idem para scheduled”

Aislar el SUT

- Las distintas funcionalidades del SUT en muchos casos dependen entre sí o de componentes ajenos al SUT.
- Cuando se producen cambios en los componentes de los que depende el test, es posible que este último empiece a fallar.
- Al testear funcionalidades del SUT es preferible no depender de componentes del sistema ajenos al test.

Qué debemos hacer?

- Test Doubles (también conocidos como “mock objects”)



Mock objects

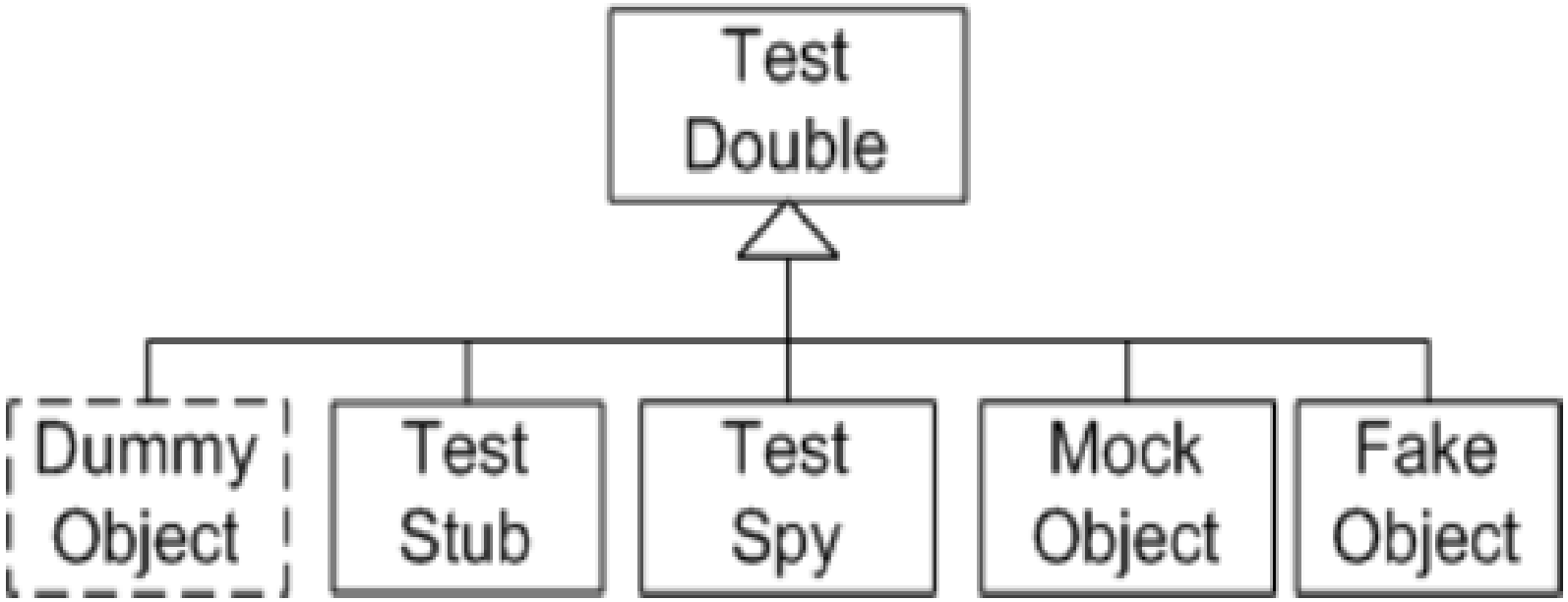
- **Mock objects** son “simuladores” que imitan el comportamiento de otros objetos de manera controlada.

Problema general

- Es necesario realizar pruebas de un “SUT” que depende de un módulo u objeto
- El módulo u objeto requerido no se puede utilizar en el ambiente de la pruebas
  - No está implementado
  - No se puede acceder
  - No se puede o es muy dificil replicar
  - Es un objeto complejo que puede tener errores en si mismo que no quiero acarrear

* SUT: System Under Testing

Tipos de test doubles



En todos los casos, TestDouble es polimórfico al objeto representado

Tipos de test doubles

- **Dummy object**: se utiliza el objeto para que ocupe un lugar pero nunca es utilizado
- **Test Stub**: sirve para que el SUT le envíe los mensajes esperados, y devuelva un valor por defecto
- **Test Spy**: Test Stub + registro de outputs
- **Mock Object**: test Stub + verification of outputs
- Fake Object: imitación. Se comporta como el módulo real (protocolos, tiempos de respuesta, etc)

Cuando usar test doubles

- Cuando el objeto real es un objeto complejo que:
  - retorna resultados no-deterministicos (ej., la hora actual o la temperatura actual).
  - tiene estados que son dificiles de reproducir (ej., un error de network);
  - es lento (ej, necesita inicializar una transaccion a la base de datos);
  - todavia no existe;
  - tiene dependencias con otros objetos y necesita ser aislado para testearlo como unidad.

Reglas del testing

- Mantener los tests independientes entre si
- Un buen test es simple, fácil de escribir y mantener (que requiera mínimo mantenimiento a medida que el sistema evoluciona)
- El objetivo de testear es encontrar bugs
- Limitaciones del testing:
  - no encuentra todos los errores
  - no puede comprobar la ausencia de errores

Mock Frameworks

- Para Pharo:
  - Mocketry https://github.com/dionisiydk/Mocketry
- Para Java:
  - Mockito
  - JMock
  - EasyMock

Bibliografia

- Kent Beck. “Simple Smalltalk Testing: with Patterns” https://web.archive.org/web/20150315073817/http://www.xprogramming.com/testfram.htm
- “xUnit Test Patterns: Refactoring Test Code”. Gerard Meszaros. http://xunitpatterns.com/index.html
- “Test Driven Development by Example”. by Kent Beck.
- “The Little Mocker”. Robert Martin. https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html
- Video: Google+ talk con Kent Beck y Martin Fowler sobre los problemas del TDD: https://www.youtube.com/watch?v=z9quxZsLcfo

Testing con BBDD

- Es mejor testear sin usar BBDD:
  - *Fake Database*: es un tipo de test double. Se crea usando *Fresh Fixture.*
  - Se usa para testear la lógica de negocio por separado
- Cuando hace falta testear la BBDD:
  - *Database Sandbox:* una copia para cada desarrollador evita el *Test Run War*
  - Es importante testear la capa de acceso a los datos
  - JUnit tiene extensiones para BBDD (DbUnit) que por ej. permite facilmente comparar un resultado de un join con una tabla plana en XML

Testing de aplicaciones web

- Que testear
  - Funcionalidad
  - Usabilidad
  - Interface con el servidor y la BBDD
  - Compatibilidad (browsers, SOs)
  - Performance
  - Securidad
- Herramientas:
  - Selenium, TestComplete, etc.
  - “You can record and play web page navigation, logging in and out, filling out and submitting forms, searching product catalogs, placing orders and other operations. You can also insert checkpoints to verify data and attributes of web page elements, link validity and the web page structure and accessibility.”