Unit testing

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

Contexto

- Nos interesa incrementar la calidad del software:
  - Funcionalidad correcta y que funcione correctamente
- Testing se puede realizar a distintos niveles:
  - Tests de aceptación, tests de integración, **tests de unidad**
- Testing efectivo asume la presencia de un framework de unit-testing (como los de la familia xUnit) que permita **automatizar** los tests:
  - Volver a ejecutar una y otra vez los tests creados
  - Visualizar el resultado fácilmente

Framework Xunit

- La(s) primera(s) letra(s) identifica el lenguaje: SUnit, JUnit, CppUnit, NUnit, PyUnit, …
- La primera herramienta de testing automático fue SUnit, escrito por Kent Beck para Smalltalk
- Por su simplicidad y funcionalidad, se llevó a prácticamente todos los lenguajes de programación, y es open source

Test de unidad (Xunit)

- Testeo de la *mínima unidad de ejecución*.
- En OOP, la mínima unidad es un método.
- **Objetivo**: aislar cada parte de un programa y mostrar que funciona correctamente.
- Cada test confirma que un método produce el output esperado ante un input conocido.
- Es como un contrato escrito de lo que esa unidad tiene que satisfacer.

Partes de un test de unidad

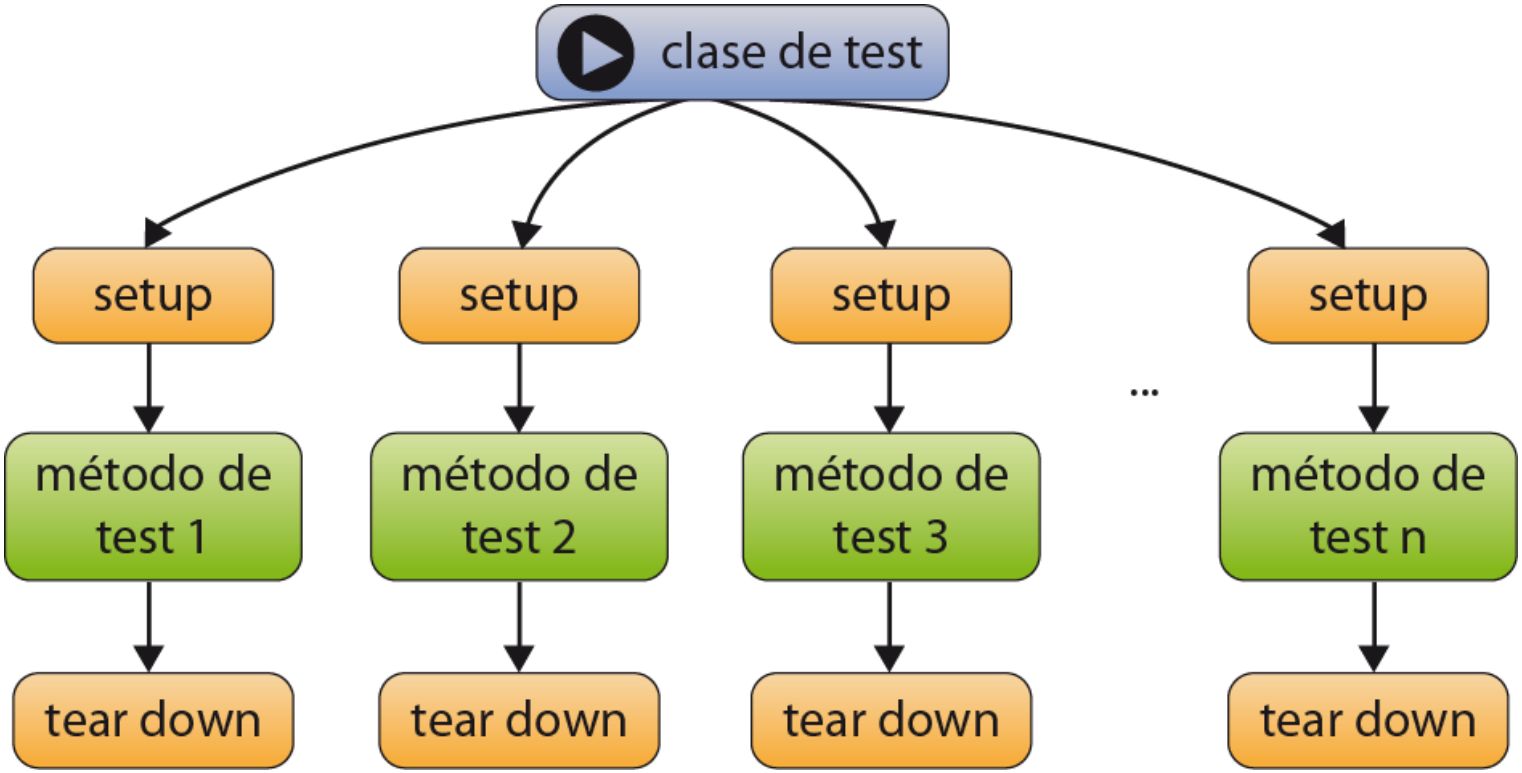
- Fase 1: Fixture set up:
  Preparar todo lo necesario para testear el comportamiento del SUT
- Fase 2: Exercise:
  Interactuar con el SUT para ejercitar el comportamiento que se intenta verificar
- Fase 3: Check:
  Comprobar si los resultados obtenidos son los esperados, es decir si el test tuvo éxito o falló
- Fase 4: Tear down
  Limpiar los objetos creados para y durante la ejecución del test

SUT: System Under Test

SUnit

- Clase TestCase que tenemos que subclasificar
- Redefinir los métodos setUp (donde se crea el fixture), tearDown (donde se borran los objetos creados) y escribir métodos de testing que empiecen con la palabra “test"

Tests con xUnit



Ejemplo

n! = 1 * 2 * … * (n – 1) * n

```
Integer>>factorial
   self < 0
      ifTrue: [^self error: 'Function out of range'].
   ^self = 1
      ifTrue: [1]
      ifFalse: [self * (self - 1) factorial]
```

Subclase de TestCase

- Subclase de TestCase: #TestInteger
- Variables de instancia : ‘zero small big neg'

setUp y tearDown

setUp

```
zero := 0.
small := 2.
big := 10.
neg := -1
```

tearDown

Mars Global Surveyor

Casos de testing

testFactorial

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

Integer>>factorial

```
self < 0
   ifTrue: [^self error: 'Function out of range'].
self = 1
   ifTrue: [1]
   ifFalse: [self * (self - 1) factorial]
```

Consideraciones

- should: o assert: ?
- ¿Qué valores testear?

should: o assert: ?

- should: aBlock
- should: aBlock raise: anException <- más usado así

- assert: aBoolean
- assert: actual equals: expected

¿Qué valores testear?

- Escribir casos de testing es deseable pero es costoso
- Testear todos los valores no es práctico
- Se busca encontrar casos importantes: aquellos donde es más probable que cometamos errores:
  - Particiones Equivalentes
  - Valores de Borde

Particiones Equivalentes

- Tratar conjuntos de datos como el mismo (si un test pasa, otros similares pasarán)
- Para rangos, elegir un valor en el rango, y un valor fuera de cada extremo del rango.
- `Debe aceptar años entre 1-2050.`
  - `Casos de testing: 0, 1876 , 2076.`
- Para conjuntos, elegir uno en el conjunto, uno fuera del conjunto.
- `Passwords deben contener un caracter numérico:`
  - `Casos de testing: a5, ab`

Valores de Borde

- La mayoría de los errores ocurren en los bordes o límites entre conjuntos
- `Debe aceptar años entre 1-2050.`
  - `Casos de testing: 0, 1, 2050 , 2051.`
- `Passwords deben ser de 6-8 caracteres de largo:`
  - `Casos de testing: abcde, abcdef, abcdefgh, abcdefghi`
- Los “Valores de Borde” complementa “Particiones Equivalentes”.

Volviendo a Integer>>factorial

```
Integer>>factorial
   self < 0
      ifTrue: [^self error: 'Function out of range'].
   ^self = 0
      ifTrue: [1]
      ifFalse: [self * (self - 1) factorial]
```

Cuándo/Cómo/Por qué testear

- “Test with a purpose” (Kent Beck)
- Saber por qué se testea algo y a qué nivel debe testearse.
- El objetivo de testear es encontrar bugs
- Se puede aplicar a cualquier artefacto del desarrollo
- Se debe testear temprano y frecuentemente
- Testear tanto como sea el **riesgo** del artefacto
- Un test vale más que la opinión de muchos