Tests de unidad

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.
- Escritos desde la perspectiva del programador
- 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.

Test de unidad – Pros y contras

- ☑ Facilita cambios
- ☑ Simplifica la integración
- ☑ Documenta cada unidad

- ☒ No encuentra todos los errores
- ☒ Puede no encontrar la ausencia de errores
- ☒ Puede no encontrar problemas de integración o de performance.

Partes de un test de unidad

- Fase 1: Fixture set up:

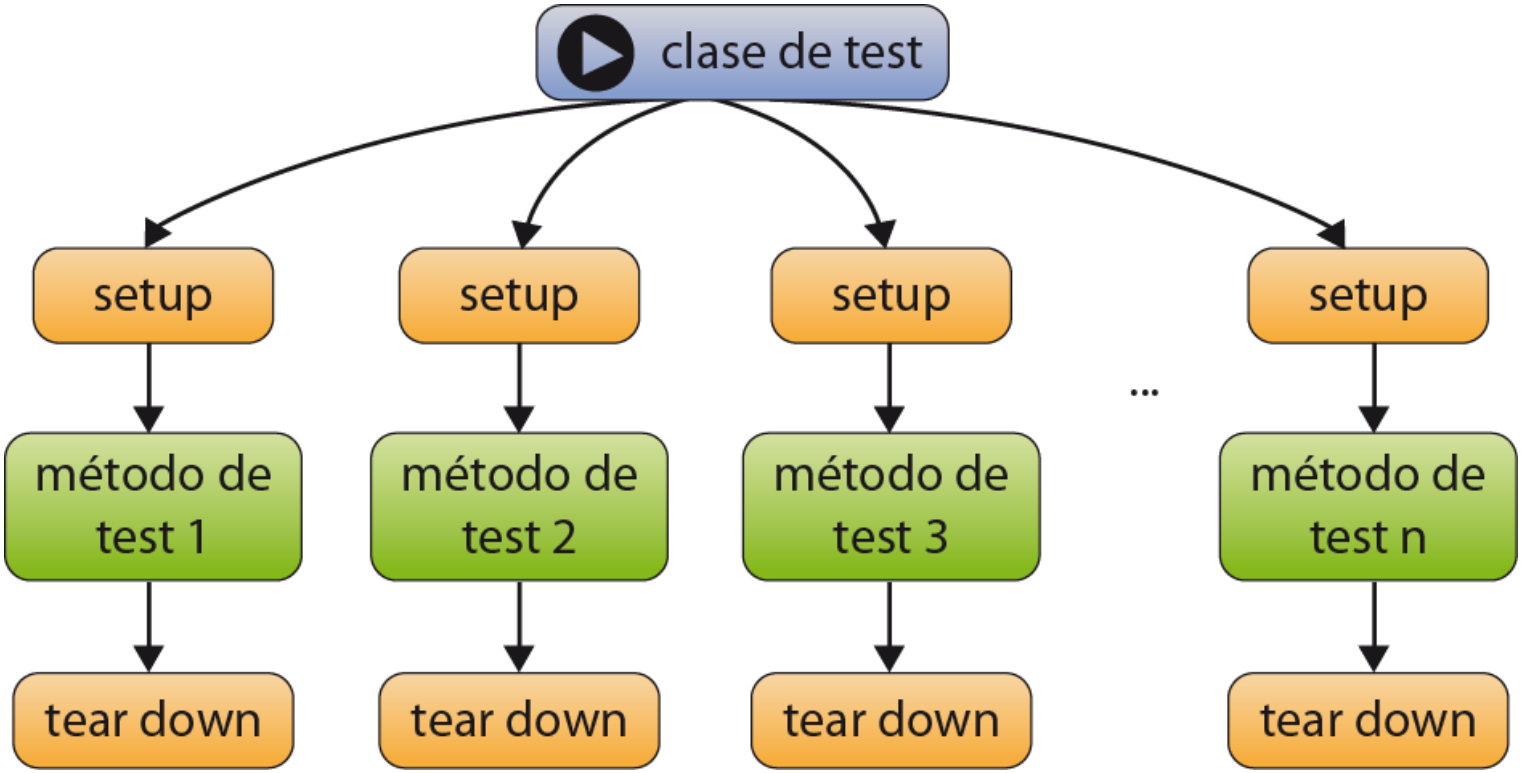
  Preparar todo lo necesario para testear el comportamiento del SUT (System Under Testing)
- Fase 2: Exercise:

  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

Tests independientes



Aislar el SUT

- Las distintas funcionalidades del SUT (System Under Testing) 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.

Mock objects

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

- Por ejemplo, un reloj alarma que debe hacer sonar una campana a determinada hora.
  Para testear el test debería esperar la hora de alarma → se usa un mock object que provee la hora de alarma.

Cuando usar Mock objects

- 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.

Ejemplo

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

Subclase de TestCase

```
Smalltalk defineClass: #TestInteger
   superclass: #{Smalltalk.SUnit.TestCase}
   indexedType: #none
   private: false
   instanceVariableNames: 'zero small big '
   classInstanceVariableNames: ''
   imports: ''
   category: 'SUnit'
```

setUp y tearDown

setUp

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

tearDown

Mars Global Surveyor

Casos de testing

```
testFactorial

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

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

should: o assert: ?

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

- assert: aBoolean
- assert: actual equals: expected

Fixture setup patterns

- 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
- Existen diferentes alternativas para organizar el *fixture setup:*
  - In-line Setup (inlined en el método de test)
  - Delegated Setup (extrayendo el inlined setup)
  - Implicit Setup (en el método setUp)
  - Hybrid Setup (combinación de los tres anteriores)

```
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"
```

Mantener los test independientes

- Mientras mayor sea la dependencia entre los test, menos exacta será información acerca de un fallo en particular.
- Si tenemos tests que dependen de la ejecución de otros, los cambios introducidos en estos últimos afectarán el comportamiento de los primeros.
- Al hacer que los tests sean independientes de la ejecución de otros, los fallos indicarán información mucho más útil y los test serán más confiables.

Tamaño de los métodos de testing

- La postura más purista consiste en 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
- Debemos aislar cada camino de ejecución y escribir un método de test que verifiquen las condiciones necesarias para testear ese camino.

- Los test con varias condiciones surgen para evitar las repetidas configuraciones del estado inicial de un test.