

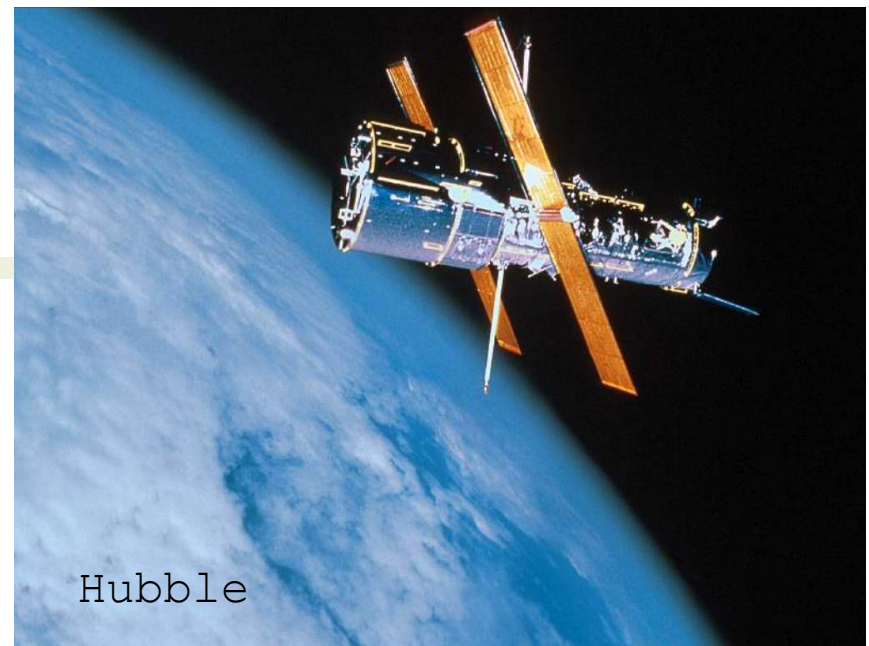
A decorative graphic consisting of a thin yellow circle on the left and a horizontal bar with a yellow-to-white gradient on the right. The text 'Testing de unidad' is centered within the bar.

# Testing de unidad

Dra. Alejandra Garrido

Objetos 2 – Fac. De Informática – U.N.L.P.

[alejandra.garrido@lifa.info.unlp.edu.ar](mailto:alejandra.garrido@lifa.info.unlp.edu.ar)



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

# [Contexto]

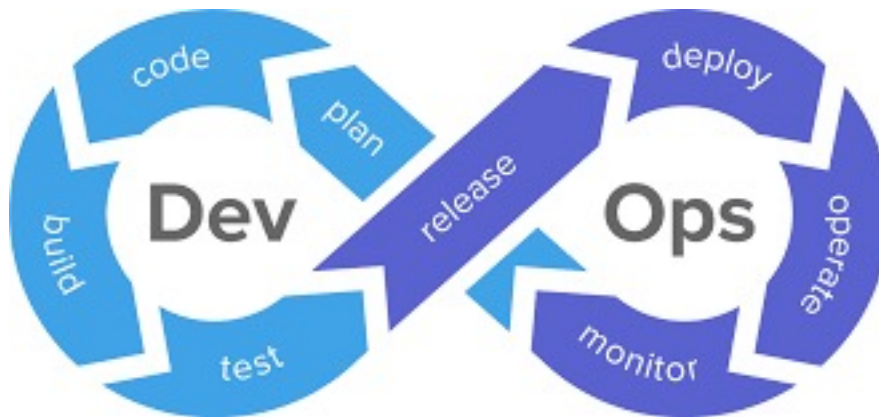
- Nos interesa incrementar la calidad del software:
  - Funcionalidad correcta y que funcione correctamente
- Testing se puede realizar a distintos niveles:
  - Tests de aceptación, tests de integración, **tests de unidad**

# [ Test de unidad ]

- 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.

# [Automatización del unit testing]

- 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



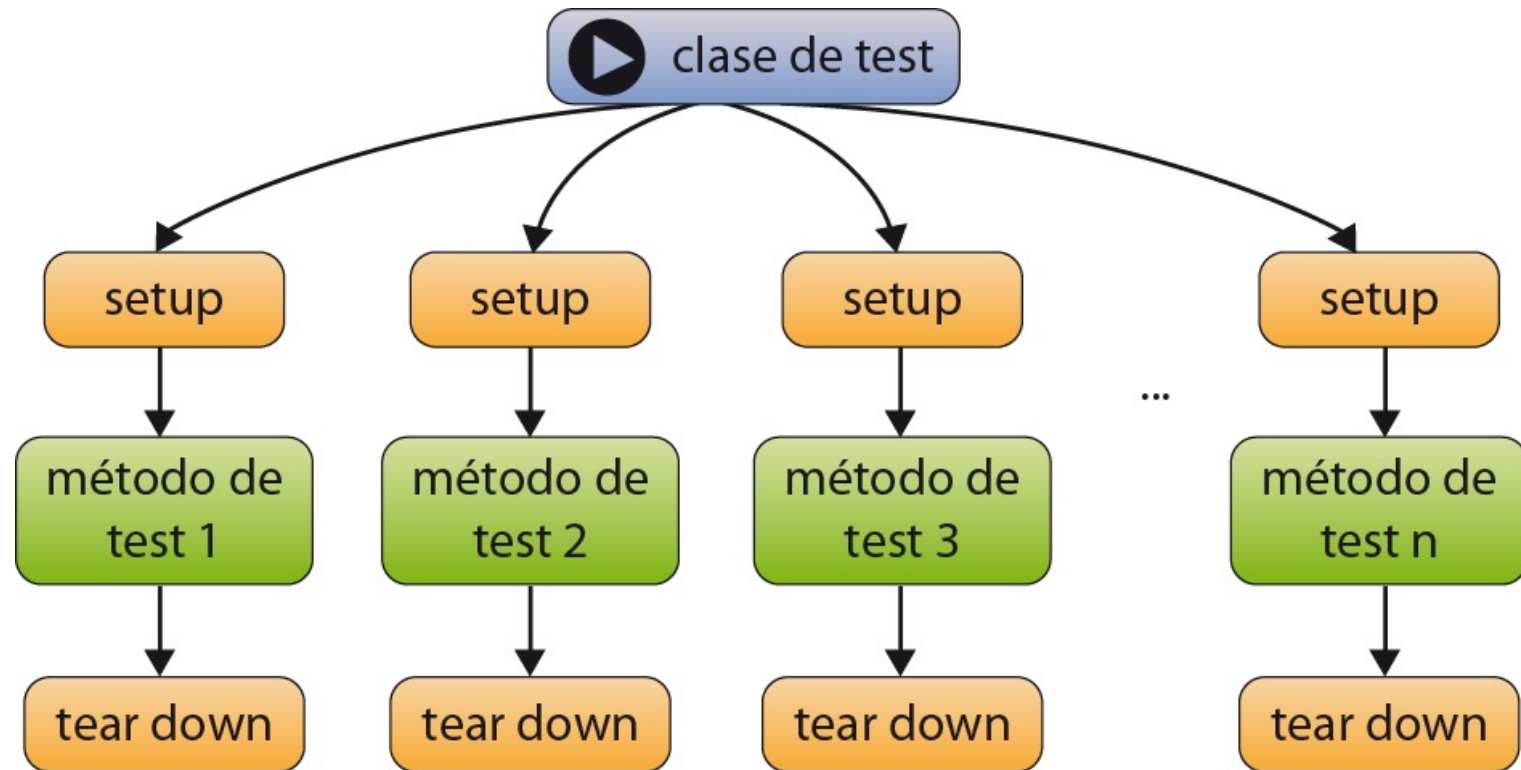
**Aprende XUnit**

# [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

# [Tests con xUnit]





Importamos las partes  
de JUnit que necesitamos

```
package ar.edu.unlp.info.oo1.ejemploTeoriaTesting;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

Definición y preparación  
del "fixture)

```
public class RobotTest {

    private Robot robot;

    @BeforeEach
    public void setUp() {
        robot = new Robot(0,100);
    }
```

Ejercitar los objetos  
Verificar resultados

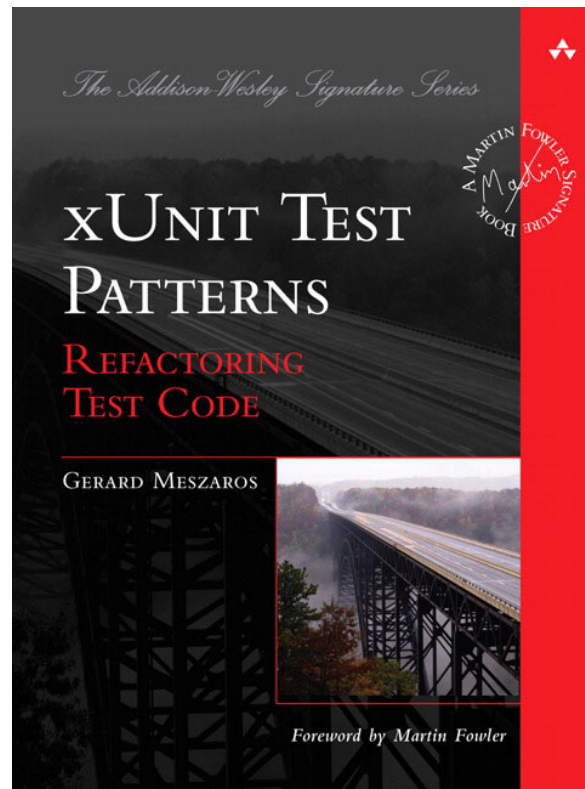
```
@Test
public void testAvanzar() {
    robot.avanzar();
    assertEquals(99, robot.getEnergia());
    assertEquals(1, robot.getPosicion());
}

@Test
public void testRetroceder() {
    robot.retroceder();
    assertEquals(99, robot.getEnergia());
    assertEquals(-1, robot.getPosicion());
}
```

Tests

```
}
```

# [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?

# Tamaño de las clases de testing

*Cómo organizamos los test methods en testcase classes?*

## ■ Testcase class per class

- Poner todos los test methods de un SUT en una única testcase class
- **How to use it?** Creamos una clase separada por cada clase que queremos testear
- **When to use it?** Es un buen punto de partida cuando no hay muchos test methods

## ■ Testcase class per feature (por cada método)

- Agrupamos test methods basados en cada feature de la clase

## ■ Testcase class per fixture

- Agrupamos los test methods que comparten un mismo fixture

# [ 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

# [ Ejemplo: In-line setup ]

```
public class FlightStateTestCase {
```

```
@Test
```

```
public void testStatusInitial() {
```

```
    // in-line setup
```

```
    Airport departureAirport = new Airport("Buenos Aires", "EZE");
```

```
    Airport destinationAirport = new Airport("Mar del Plata", "MDQ");
```

```
    Flight flight = new Flight(flightNumber, departureAirport ,  
                                destinationAirport);
```

```
    // exercise SUT and verify outcome
```

```
    assertEquals(FlightState.PROPOSED, flight.getStatus());
```

```
}
```

# [ Ejemplo: In-line setup ]

```
public class FlightStateTestCase {
```

```
@Test
```

```
public void testStatusCancelled() {
```

```
    // in-line setup
```

```
    Airport departureAirport = new Airport("Buenos Aires", "EZE");
```

```
    Airport destinationAirport = new Airport("Mar del Plata", "MDQ");
```

```
    Flight flight = new Flight(flightNumber, departureAirport ,  
                                destinationAirport);
```

```
    flight.cancel();
```

```
    // exercise SUT and verify outcome
```

```
    assertEquals(FlightState.CANCELLED, flight.getStatus());
```

```
}
```

# [ Fixture Setup Patterns ]

## ■ In-line Setup

- Cada test method crea su propio fixture nuevo

## ■ Delegated Setup

- Cada test method crea su propio fixture llamando a Métodos de Creación
- `createAnonymousFlight();`
- `createAnonymousCancelledFlight();`

## ■ Implicit Setup

- Se construye un test fixture común a varios tests en un método `setUp()`



# [ Aislar el SUT ]

- Dijimos que el fixture también debe “crear e inicializar todo aquello de lo que el SUT depende o que le va a ser pasado como argumento”
- 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 ]

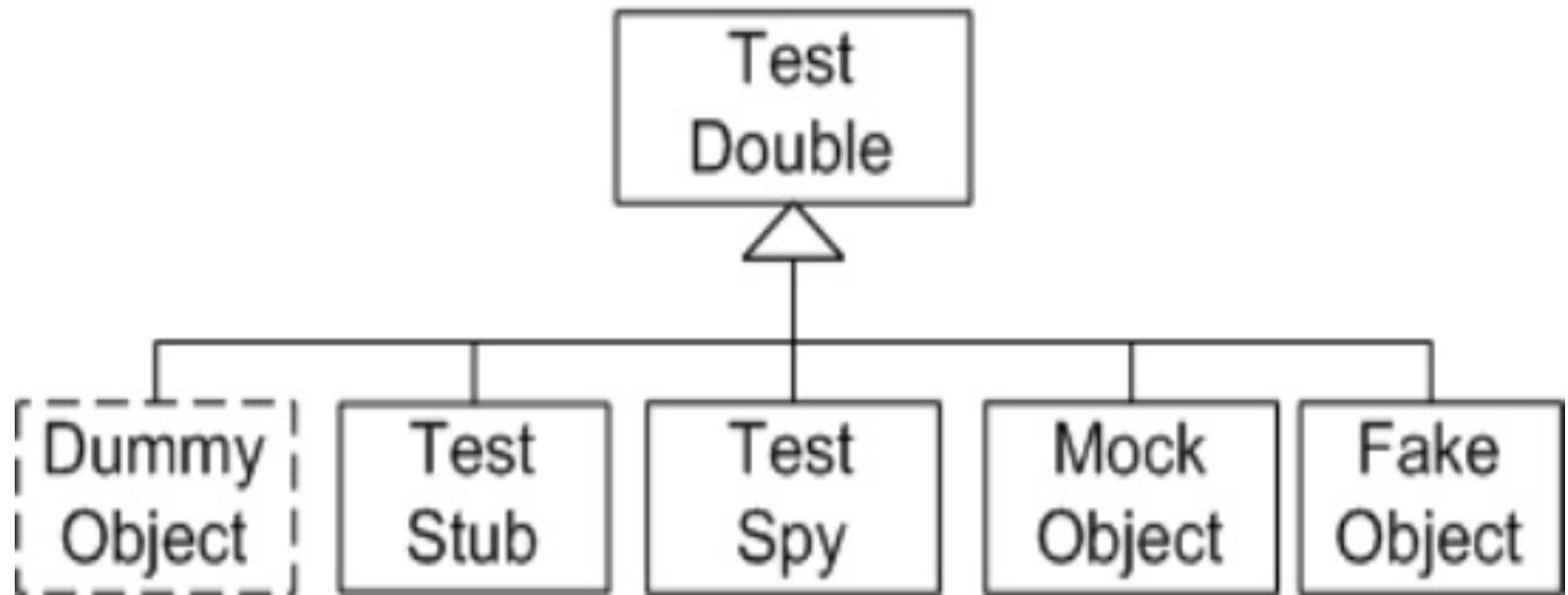
- **Mock objects** son “simuladores” que imitan el comportamiento de otros objetos de manera controlada.



# [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 difícil replicar
  - Es un objeto complejo que puede tener errores en si mismo que no quiero acarrear

# [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)

# [ Ejemplo de test stub ]

```
public void testDisplayCurrentTime_AtMidnight() throws Exception {  
    // Fixture setup  
    //    Test Double configuration  
    TimeProviderTestStub tpStub = new TimeProviderTestStub();  
    tpStub.setHours(0);  
    tpStub.setMinutes(0);  
    //    Instantiate SUT  
    TimeDisplay sut = new TimeDisplay();  
    sut.setTimeProvider(tpStub);  
    // Exercise SUT  
    String result = sut.getCurrentTimeAsHtmlFragment();  
    // Verify outcome  
    String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";  
    assertEquals("Midnight", expectedTimeString, result);  
}
```

# [ 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.



# [ Java Mocking Frameworks ]

- Mockito, EasyMock, Jmockit

```
// Ejemplo en Mockito: https://site.mockito.org  
LinkedList mockedList = mock(LinkedList.class);
```

```
// stubbing  
when(mockedList.get(0)).thenReturn("first");
```

```
// usando el stub; muestra "first"  
System.out.println(mockedList.get(0));
```

```
// muestra "null" porque get(999) no fue definido para el stub  
System.out.println(mockedList.get(999));
```

# [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 (se habla de % de cobertura)

# [ Bibliografia ]

- “xUnit Test Patterns: Refactoring Test Code”. Gerard Meszaros. <http://xunitpatterns.com/index.html>
- “Test Driven Development by Example”. by Kent Beck.
- “The Little Mockers”. Robert Martin.  
<https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMockers.html>
- Video: Google+ talk con Kent Beck y Martin Fowler sobre los problemas del TDD:  
<https://www.youtube.com/watch?v=z9quxZsLcfo>