PRUEBAS UNITARIAS

PROGRAMACIÓN III

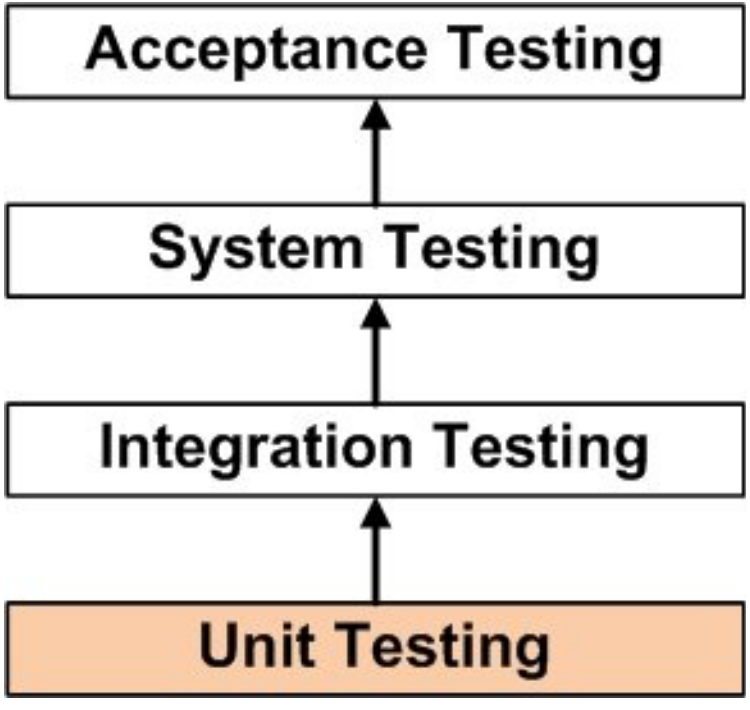
Abdel G. Martínez L.

INTRODUCCIÓN

- Una aplicación es la suma de cada una de sus partes.
- Como un todo, las aplicaciones entregan potencialmente una gran cantidad de valor a las organizaciones.
- Si uno de sus componentes no funciona como esperado, entonces la aplicación completa falla. Así nace la prueba unitaria.
- En esa presentación se explica cómo crear una prueba unitaria, sus componentes básicos y cuáles son los lineamientos a seguir en Python para realizar las pruebas.

¿QUE ÉS UNA PRUEBA UNITARIA?

- Se cubre el nivel más básico de la aplicación.
- Se prueba cada unidad individual de código.
- Se aísla el método para ver si dada ciertas condiciones previamente establecidas, la aplicación responde de la manera esperada.
- Detallar las pruebas a este nivel permite tener la confianza de que cada parte de la aplicación funcionará como esperado.

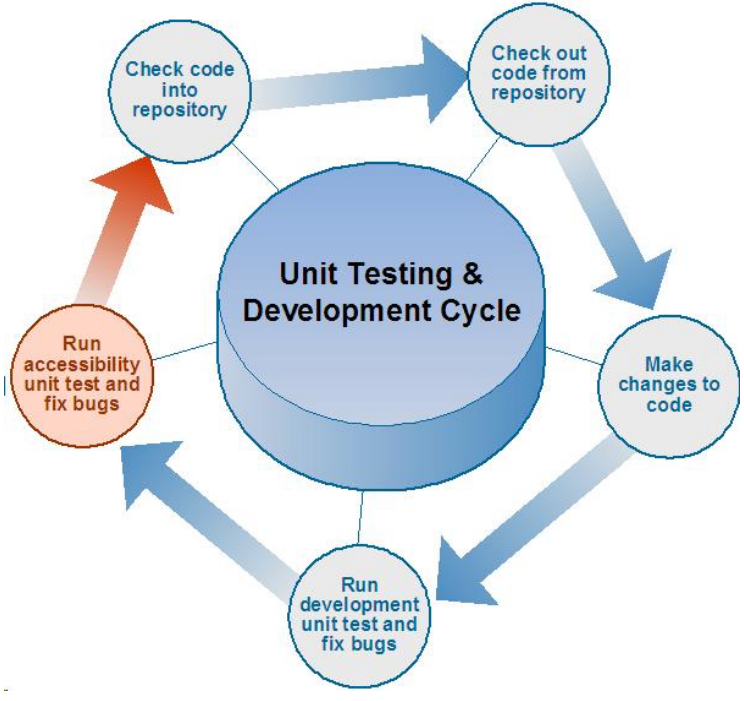


CARACTERÍSTICAS

Automatizable	Completas	Repetibles	Independientes	Profesionales
• No debería requerir una intervención manual.	• Deben cubrir la mayor cantidad de código.	• No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.	• La ejecución de una prueba no debe afectar a la ejecución de otra.	• Las pruebas deben ser consideradas igual que el código, con la misma documentación.

¿QUÉ SE DEBE PROBAR?

- Se debe enfocar en pequeñas unidades de código del proyecto.
- Las pruebas deben ser lo más flexibles posibles para contemplar mejoras sobre las funcionalidades.
- Para encontrar bugs, de una aplicación en cualquier ambiente.



VENTAJAS

Fomentan el cambio

- Facilita que el programador cambie el código para mejorar su estructura.

Simplifica la integración

- Permiten llegar a la fase de intergración con un grado alto de seguridad de que el código funciona.

Documenta el código

- Las propias pruebas son documentación del código puesto que ahí se puede ver como utilizarlo.

Separa la interfaz y la implementación

- Se pueden utilizar objectos mock para simular el comportamiento de objetos complejos.

Los errores son más fáciles de ubicar

- Las pruebas unitarias pueden desenmascararlos.

DESVENTAJAS

- No descubren todos los errores del código.
- Testing Aleatorio: técnica de eneración aleatoria de objetos para amplificar el alcance de las pruebas de unidad.
- No descubren errores de integración, problemas de rendimiento y otros problemas que pueden afectar al sistema en su conjunto.
- Las pruebas unitarias sólo son efectivas si son utilizadas en conjunto con otras pruebas de software.

ESTÁNDAR PEP-8

- **Identación**: Cuatro espacios por cada identación.
- **Longitud de línea máxima**: 80 caracteres.
- **Líneas en blanco**: Dos entre la definición de importaciones, clases y funciones. Una entre definición de métodos dentro de una clase.
- **Sentencias de importación**: Debe ser una por línea
- **Nombre de clases**: Debe tener en mayúscula la primera letra de cada palabra que la componga.
- **Nombre de métodos**: Debe usar minúsculas y guiones bajos para separar palabras.

ESTRUCTURA DE PRUEBA UNITARIA

- Las pruebas unitarias deben ubicarse en un directorio llamado test/unit al nivel más superior del proyecto.
- Todos los directorios dentro del código de la aplicación deben tener directorios espejo dentro de test/unit.
- Todos los ficheros de pruebas unitarias deben tener el mismo nombre cuando se están probando, incluyendo el sufijo _test.

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- **`assertEqual(x,y,msg=None)`**
- Este método valida si el argumento x es igual al argumento y.
- Por debajo utiliza un == para validar la definición de los objetos.

```
def test_assert_equal(self):
    self.assertEqual(1, 1)
```

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- **`assertAlmostEqual(x,y,places=None,delta=None)`**
- Método útil cuando se hacen cálculos que deben dar un resultado que esté dentro de cierta cantidad de cifras o de cierto delta.

```
def test_assert_almost_equal_delta_0_5(self):
    self.assertAlmostEqual(1, 1.2, delta=0.5)
def test_assert_almost_equal_places(self):
    self.assertAlmostEqual(1, 1.00001, places=4)
```

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- **`assertRaises(exception,method,arguments,msg=None)`**
- Método para validar que un método y un conjunto de argumentos levanten una determinada excepción.

```
def test_assert_raises(self):
    self.assertRaises(ValueError, int, ”a”)
```

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- **assertDictContainsSubset(expected,actual,msg=None)**
- Método para validar que actual contenga a expected. Útil para validar parte de un diccionario está presente en el resultado.

```
def test_assert_dict_contains_subset(self):
    expected = {'a': 'b'}
    actual = {'a': 'b', 'c': 'd', 'e': 'f'}
    self.assertDictContainsSubset(expected, actual)
```

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- **`assertDictEquals(d1,d2,msg=None)`**
- Método que valida que dos diccionarios contengan exactamente el mismo pair de llaves y valores. Pueden estar en desorden.

```
def test_assert_dict_equal(self):
    expected = {'a': 'b', 'c': 'd'}
    actual = {'c': 'd', 'a': 'b'}
    self.assertDictEqual(expected, actual)
```

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- `assertTrue(expr,msg=None)`
- `assertFalse(expr,msg=None)`
- `assertGreater(a,b,msg=None)`
- `assertGreaterEqual(a,b,msg=None)`
- `assertIn(member,container,msg=None)`
- `assertIs(expr1,expr2)`
- `assertIsInstance(obj,class,msg=None)`
- `assertNotIsInstance(obj,class,msg=None)`

MÉTODOS ÚTILES EN PRUEBA UNITARIA

- `assertIsNone(obj,msg=None)`
- `assertIsNot(expr1,expr2,msg=None)`
- `assertIsNotNone(obj,msg=None)`
- `assertLess(a,b,msg=None)`
- `assertLessEqual(a,b,msg=None)`
- `assertItemsEqual(a,b,msg=None)`