ENTORNOS DE DESARROLLO Y TEST UNITARIOS

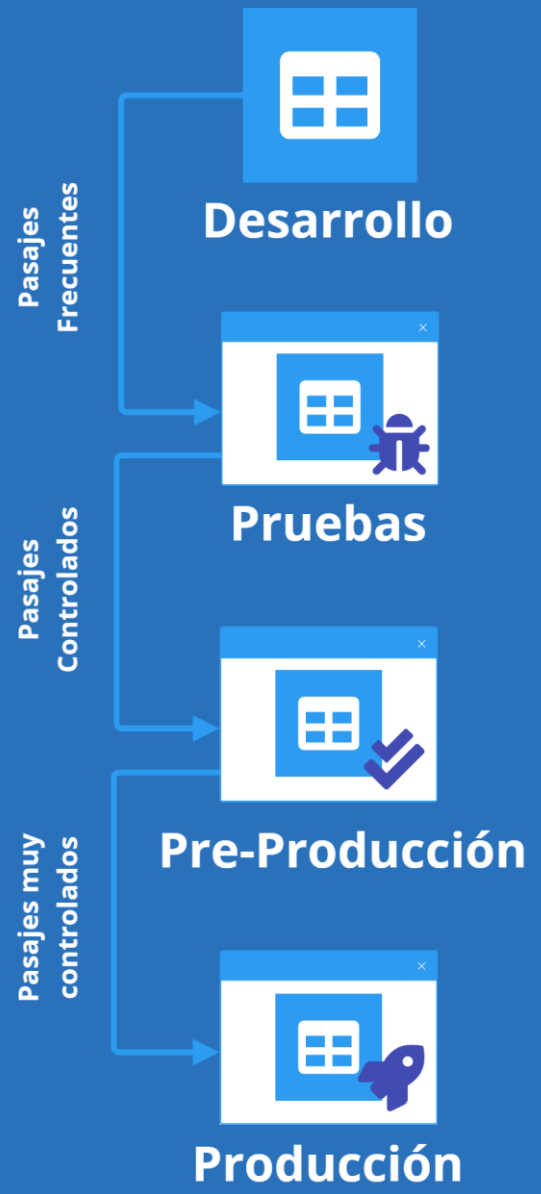
Algoritmos y estructuras de datos I

Lic. Yaps David P.

ENTORNOS DE TRABAJO EN EL DESARROLLO

Para desarrollar una aplicación, se necesitan al menos dos entornos: desarrollo y producción. El entorno de desarrollo es donde el programador escribe y prueba el código, mientras que el de producción es donde la aplicación se despliega y se hace pública.

A medida que el proyecto avanza, pueden añadirse más entornos, como pruebas, para que el cliente valide funcionalidades, y preproducción, que el equipo de QA usa para asegurar la calidad y detectar errores antes del despliegue final.



Lic. Yaps David P.

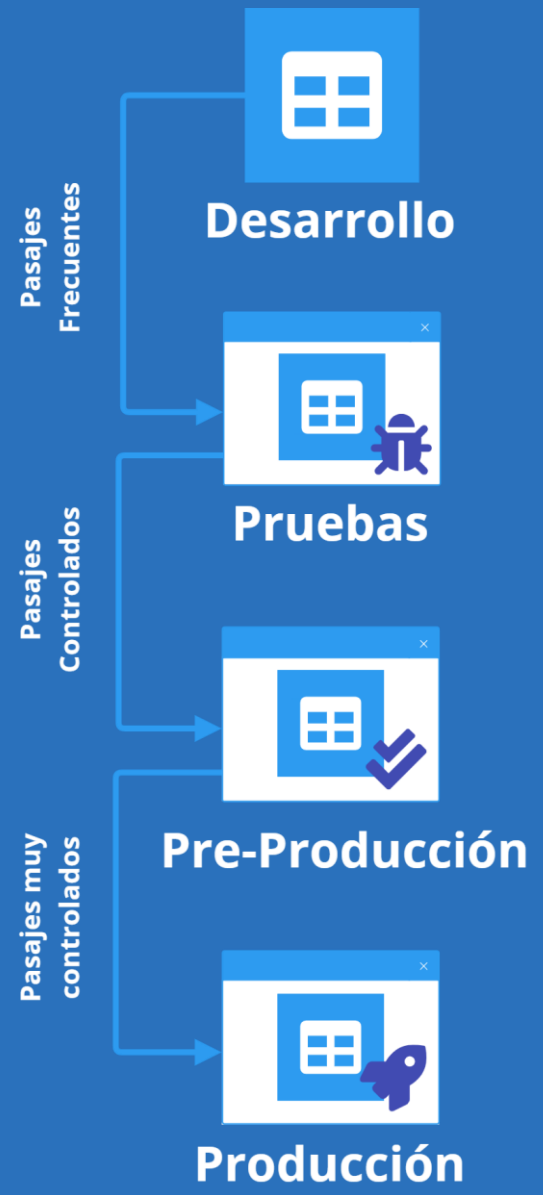
ENTORNOS DE TRABAJO EN EL DESARROLLO

Desarrollo

Es el espacio donde los programadores crean y prueban el código en las etapas iniciales de un proyecto. Se encuentra en sus equipos y ofrece herramientas de depuración que agilizan la resolución de errores. Soporta iteraciones rápidas, realizar pruebas unitarias y hacer ajustes en tiempo real, lo cual acelera el desarrollo y facilita la integración continua.

Pruebas

El entorno de pruebas verifica la funcionalidad del software y detecta errores antes de etapas críticas. Usado por *testers* y, a veces, por clientes, simula condiciones reales para realizar pruebas funcionales y no funcionales. Se emplean datos de prueba y herramientas de automatización para asegurar cobertura y eficiencia en el proceso de testing.



Lic. Yaps David P.

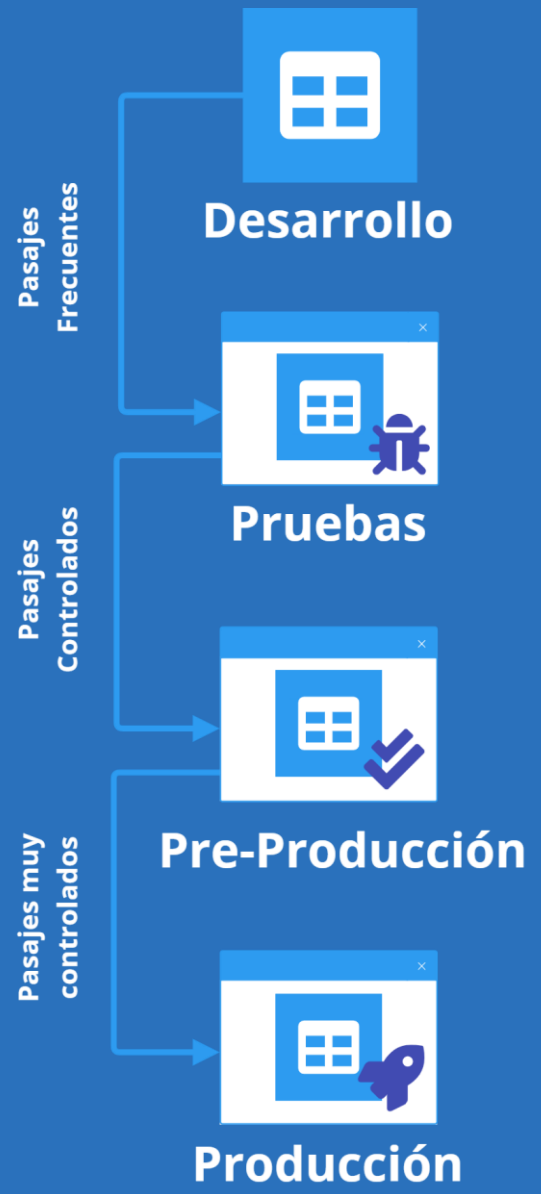
ENTORNOS DE TRABAJO EN EL DESARROLLO

Pre-producción

Es una réplica del entorno de producción, donde el equipo de QA realiza las pruebas finales para asegurar que el software cumpla con los estándares de calidad y esté libre de errores. Es la última etapa de validación antes del despliegue público.

Producción

El entorno de producción es donde la aplicación se despliega para los usuarios finales. Debe ser muy estable y seguro, ya que cualquier problema afecta a los usuarios reales. En este entorno, se gestionan datos reales y se monitorea continuamente para asegurar su disponibilidad y rendimiento óptimos.



¿POR QUÉ HACER PRUEBAS?

Existen diversos tipos de pruebas de software que podemos emplear para asegurar que nuestro software sigue funcionando correctamente después de introducir cambios en el código fuente.

Es esencial no solo verificar que nuestros usuarios puedan utilizar la aplicación (por ejemplo, iniciar sesión, enviar mensajes o actualizar datos), sino también comprobar que el sistema funcione adecuadamente cuando se ingresen datos incorrectos o se realicen acciones inesperadas.

Un buen conjunto de pruebas debe esforzarse por «romper» la aplicación, ayudándonos a comprender sus límites y a identificar áreas susceptibles de mejora.

PRUEBAS MANUALES Y AUTOMATIZADAS

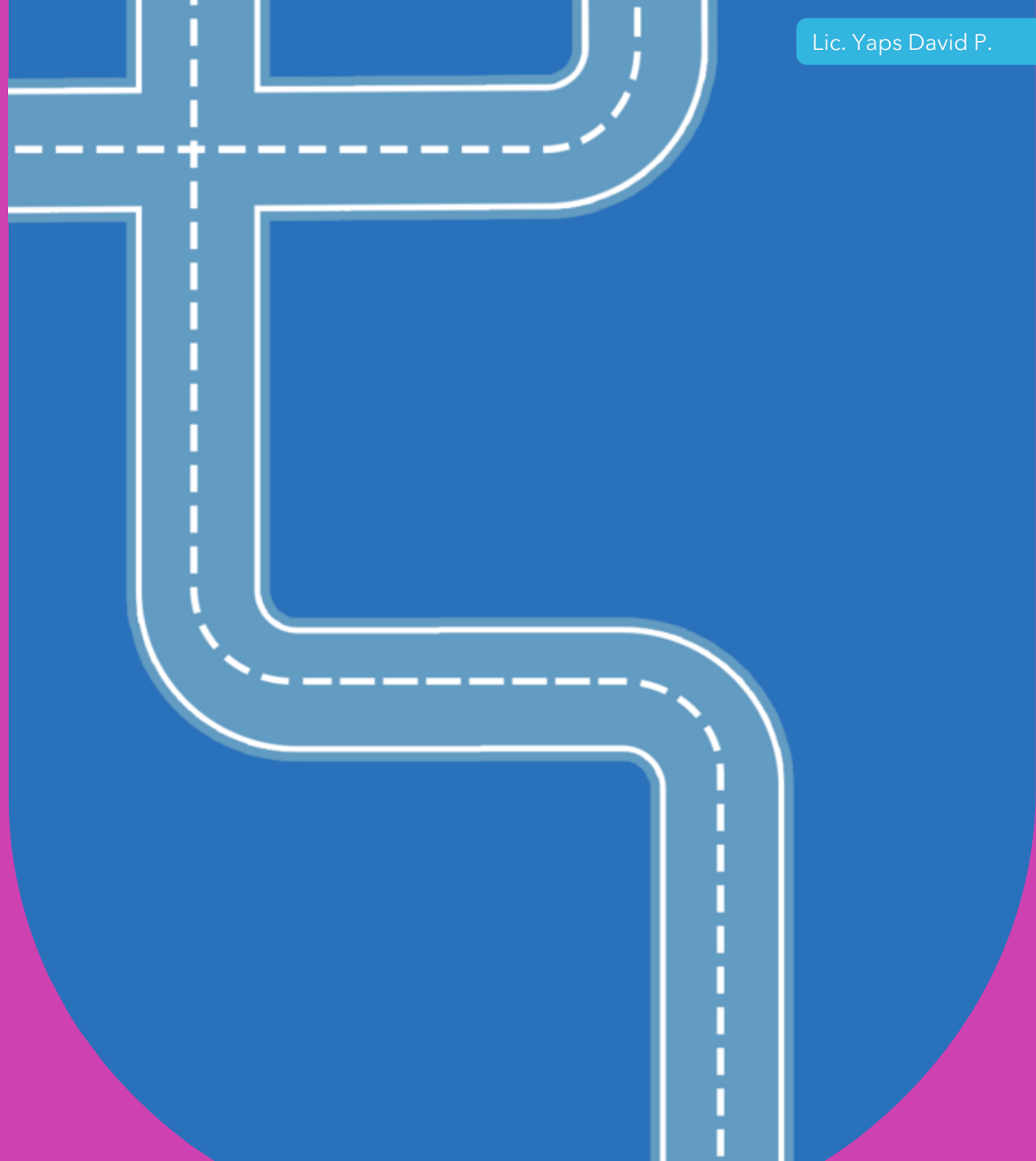
Lic. Yaps David P.

Pruebas Manuales

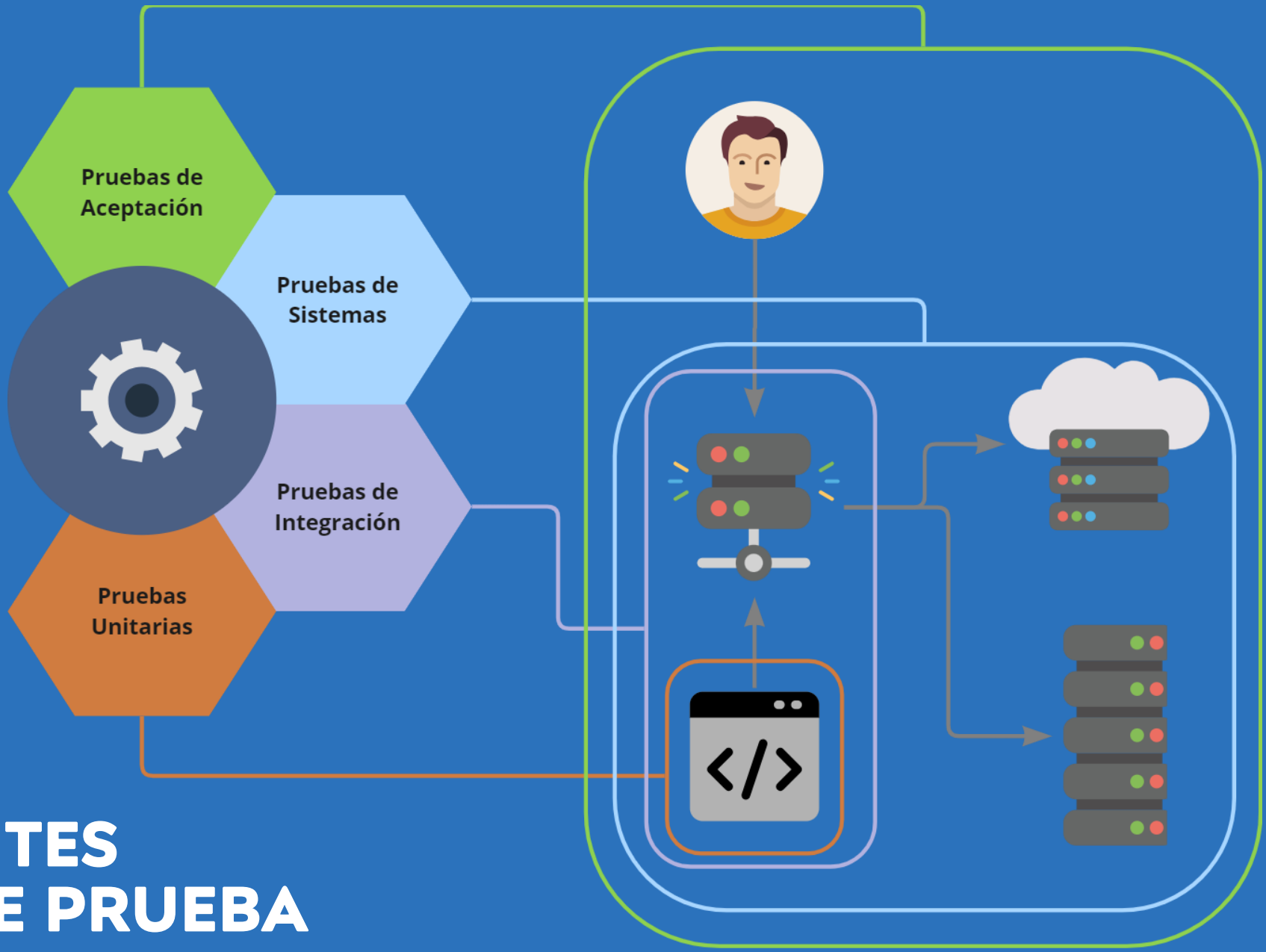
Son llevadas a cabo por personas, quienes navegan e interactúan con el software utilizando herramientas adecuadas para cada caso. Estas pruebas son costosas, ya que requieren profesionales dedicados para configurar el entorno y ejecutar las pruebas. Además, están expuestas a errores humanos, como errores tipográficos u omisión de pasos durante la prueba.

Pruebas Automatizadas

Son realizadas por máquinas que ejecutan scripts de prueba previamente escritos. Estos pueden variar en complejidad, según la necesidad del programa. Las pruebas automatizadas son más rápidas y confiables que las manuales, aunque su efectividad depende de la calidad con la que se hayan escrito los scripts de prueba.

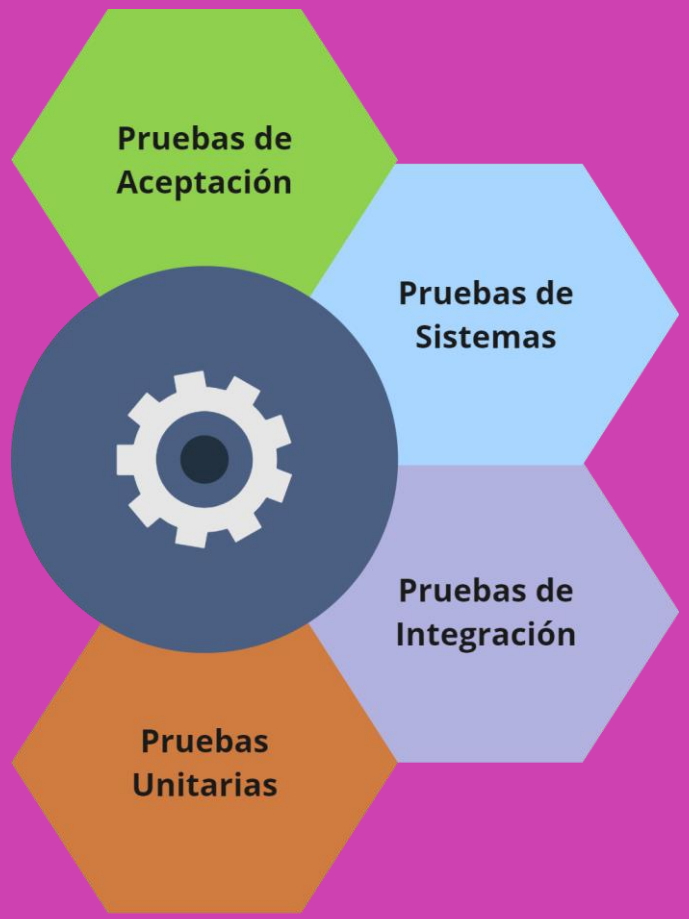


Lic. Yaps David P.



DIFERENTES TIPOS DE PRUEBA

Lic. Yaps David P.



DIFERENTES TIPOS DE PRUEBA

Pruebas Unitarias

Validan individualmente las unidades más pequeñas del código, como las funciones, asegurando que cada una actúe correctamente de forma aislada.

Pruebas de Integración

Verifican la interacción entre múltiples componentes o módulos del software, asegurando que funcionen bien juntos.

Pruebas de Sistemas

Evalúan el sistema completo desde el inicio hasta el final, simulando escenarios reales para asegurar que todas las partes del sistema funcionen correctamente en conjunto.

Pruebas de Aceptación

Confirman que el software cumple con los requisitos y expectativas del usuario final, asegurando que el producto es funcional y adecuado para su uso.

ASSERT

La responsabilidad de un desarrollador no finaliza solo en la escritura del código. Es muy importante realizar pruebas del software para asegurarse de que está libre de errores. *Assert* es una ayuda de depuración, no un mecanismo para manejar errores en tiempo de ejecución. El objetivo de usar aserciones es permitir que los desarrolladores encuentren la causa raíz de un error más rápidamente.

Assert nos permite realizar comprobaciones. Si la expresión contenida dentro de *assert* es falsa, se lanzará una excepción *AssertionError*:

```
def suma(a, b):
    return a + b

resultado = suma(2, 2)
assert resultado == 4, "2 + 2 debería ser 4"

#AssertionError
assert resultado == 5, "2 + 2 debería ser 4"
```

Se podría conseguir el mismo resultado haciendo lo siguiente, pero el uso de la función *assert* resulta más cómodo:

```
def suma(a, b):
    return a + b

resultado = suma(2, 2)
if resultado != 4:
    raise AssertionError("2 + 2 debería ser 4")
```

Lic. Yaps David P.

PRUEBAS UNITARIAS EN PYTHON

```python
def suma (a, b):
    return a + b
```

main.py

```python
from main import suma

def test_suma():
    assert suma(11, 22) == 33

def test_suma2():
    assert suma(4, 3) == 7
    assert suma(-1, 1) == 0
    assert suma(1, -1) == 0
    assert suma(0, 0) == 0
```

test_main.py

Inicialmente, debemos instalar la librería *pytest* ejecutando el siguiente comando en la terminal. Este comando utiliza *pip*, un sistema de gestión de paquetes empleado para instalar y administrar paquetes de software escritos en Python:

```
pip install pytest
```

Para continuar, trabajaremos inicialmente con dos archivos. Cuando escribimos pruebas unitarias, la instrucción *assert* se utiliza para verificar que el valor obtenido es el esperado.

Se pueden incluir múltiples *asserts* en un solo *test*. Esto permite verificar diferentes condiciones dentro de la misma función de prueba. Sin embargo, hay que tener en cuenta que si un assert falla, el *test* se detendrá y no se ejecutarán los *asserts* restantes.

PRUEBAS UNITARIAS EN PYTHON

Para poder ejecutar las pruebas, en la ubicación en la que se encuentran los archivos que deseamos probar, escribimos en la línea de comandos:

```
pytest
```

Todos los archivos dentro de la carpeta con el prefijo *test_* lo va a reconocer como una fuente de prueba para *pytest*. Dentro de dichos archivos, también ejecutará aquellas funciones que tengan el mismo prefijo.

Y obtendremos un resultado similar al siguiente (ejecutó el archivo *test_main.py* que habíamos creado):

```
============================= test session starts =============================
platform win32 -- Python 3.10.2, pytest-7.1.2, pluggy-1.0.0
collected 1 item

test_main.py .                                                           [100%]

============================== 1 passed in 0.41s ==============================
```

PRUEBAS UNITARIAS EN PYTHON

Ahora volvemos a ejecutar el comando pytest (podemos agregarle el parámetro -v para que nos demuestra el resultado del proceso de cada ítem):

```
pytest -v
```

En donde se nos indica cada uno de los llamados y si su resultado fue correcto o no.

Y en pantalla, el resultado de la ejecución que obtendremos será similar al siguiente:

```
==================================
test session starts
==================================
platform win32 -- Python 3.10.2, pytest-7.1.2,
pluggy-1.0.0 –
collected 2 items

test_main.py::test_suma PASSED          [ 50%]
test_main.py::test_suma2 PASSED         [100%]
==================================
2 passed in 0.03s
==================================
```

PRUEBAS UNITARIAS EN PYTHON

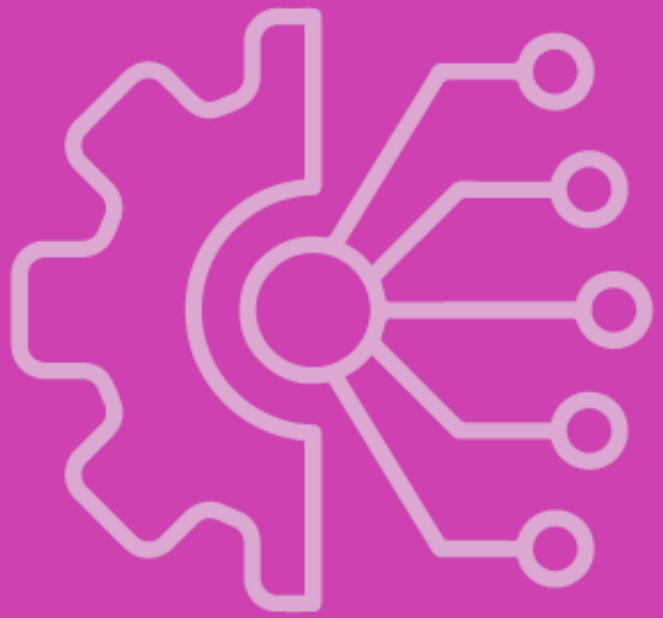
Si llegase a fallar alguno de los ítems, tendríamos un resultado similar al siguiente:

```
============================= test session starts ==============================
collected 2 items

test_main.py .F                                                          [100%]

=================================== FAILURES ===================================
_____________________________________test_suma2 _____________________________________
    def test_suma2():
>       assert suma(4, 3) == 5
E       assert 7 == 5
E        +  where 7 = suma(4, 3)

test_main.py:7: AssertionError
============================= short test summary info ==========================
FAILED test_main.py::test_suma2 - assert 7 == 5
============================= 1 failed, 1 passed in 0.18s ======================
```



INTEGRACIÓN CONTINUA

Es una práctica de desarrollo de software en la que los desarrolladores integran su código en un repositorio compartido con frecuencia, a menudo varias veces al día. Cada integración se verifica automáticamente mediante pruebas y compilaciones automáticas, lo que permite detectar y resolver errores rápidamente y asegura que el código funcione correctamente en conjunto con otros cambios.

Cada vez que un desarrollador integra nuevo código, el sistema ejecuta automáticamente una serie de pruebas unitarias para verificar que las unidades individuales de código (como funciones o módulos) actúen como se espera. Esto permite identificar y corregir errores de inmediato, evitando que fallos pequeños crezcan y afecten el proyecto en etapas posteriores.

Lic. Yaps David P.

AUTOEVALUACIÓN

La autoevaluación es crucial para identificar mejoras y fortalezas de forma objetiva, facilitando la fijación de metas y la toma de acciones para alcanzarlas.