UNIDAD DE TRABAJO 3: DISEÑO Y REALIZACIÓN DE PRUEBAS.

César San Juan Pastor

UNIDAD DE TRABAJO 3:DISEÑO Y REALIZACIÓN DE PRUEBAS.
(CONTENIDOS)

1. Introducción
2. Estrategia de pruebas: unitarias, de integración y de validación
3. Tipos de prueba: de caja blanca y caja negra
4. Herramientas de depuración de código
5. Automatización de las pruebas
6. Calidad del software (Ampliación)

INTRODUCCIÓN

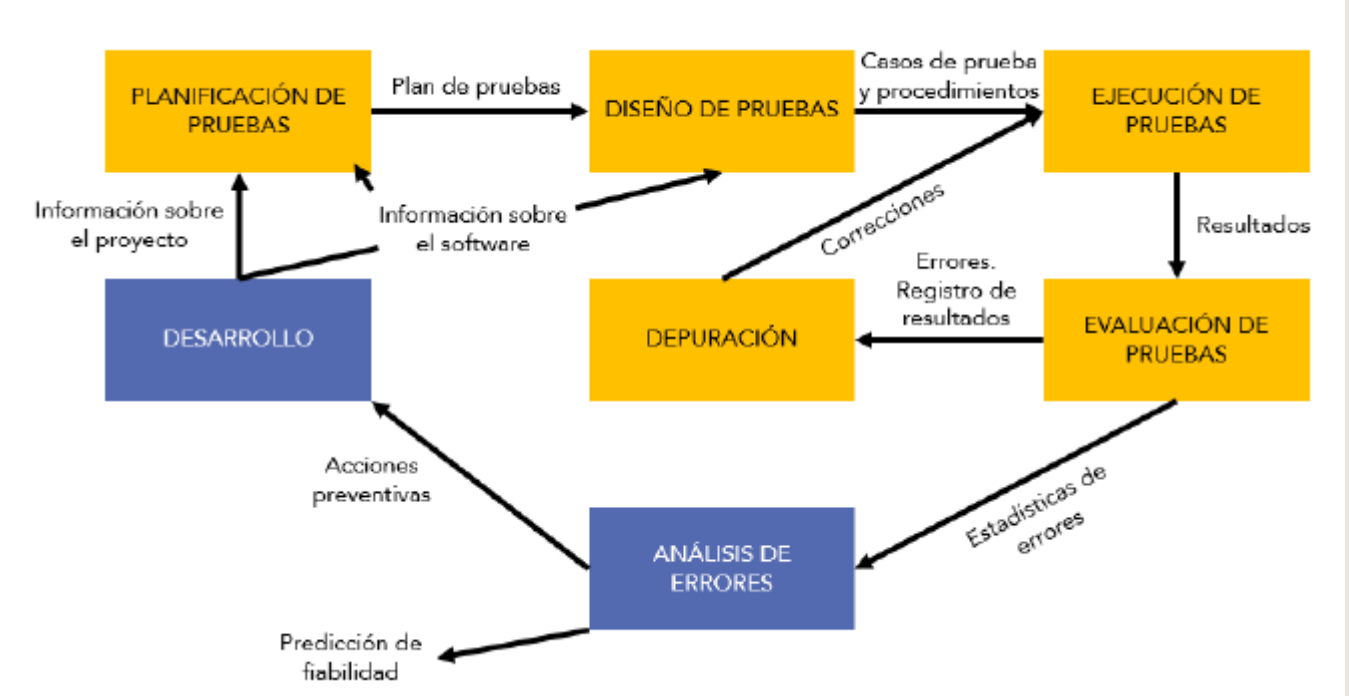
EN ESTA UNIDAD APRENDEREMOS A

- Identificar los diferentes tipos de pruebas.
- Definir casos de prueba.
- Efectuar pruebas unitarias de clases y funciones.
- Efectuar pruebas de integración, de sistema y de aceptación.
- Realizar medidas de calidad sobre el software desarrollado.

INTRODUCCIÓN

CICLO DE PRUEBAS

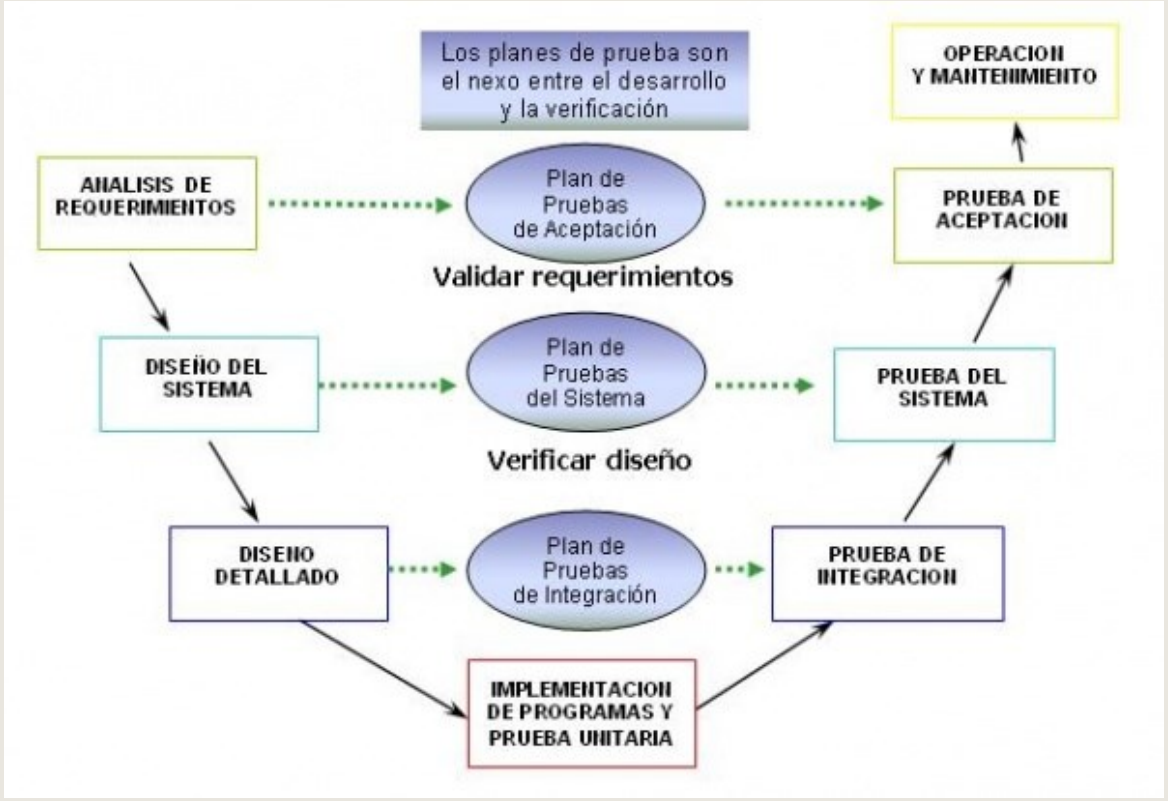
- Trataremos de encontrar errores en la codificación en la especificación o en el diseño. Durante esta fase se realizan las siguientes tareas:
  - **Verificación**: conjunto de actividades que permiten comprobar si el producto se está construyendo correctamente.
  - **Validación**: acciones para comprobar si el producto es correcto y si se ajusta a los requisitos del cliente.
- El flujo de las pruebas sería:



ESTRATEGIA DE PRUEBAS

GENERALIDADES

Las pruebas comienzan por los componentes más pequeños para avanzar de forma incremental hasta probar la aplicación completa.



ESTRATEGIA DE PRUEBAS

PRUEBAS DE UNIDAD

Comprobar cada parte de nuestro código para eliminar cualquier tipo de error en la interfaz o en la lógica interna.

- Se realizarán pruebas sobre
  - Las funciones creadas.
  - Las clases definidas.
  - Los módulos existentes.
  - La estructura de datos locales: comprobación de integridad.
  - Las condiciones límite: comprobación de que funciona en los límites establecidos.
  - Caminos independientes de la estructura de control, lo que implica asegurar de que se ejecutan las sentencias al menos una vez.
  - Todos los caminos de manejo de errores

ESTRATEGIA DE PRUEBAS

PRUEBAS DE INTEGRACIÓN

Comprobar la interacción de los distintos módulos del programa (tanto en los paradigmas estructurados como en el paradigma POO, clases y sus relaciones).

- Hay dos enfoques diferentes:
  - **Integración no incremental** o big bang. Comprobación de cada módulo por separado y después se prueba de forma conjunta. Se detectan muchos errores y la corrección es difícil.
  - **Integración incremental**. En este caso el programa se va creando y probando en pequeñas secciones por lo que localizar los fallos es más sencillo. En esta integración podemos optar por dos estrategias:
    - Ascendente. Se comienza con los módulos más bajos del programa.
    - Descendente. Se empieza por el módulo principal descendiendo por la jerarquía de control.

ESTRATEGIA DE PRUEBAS

PRUEBAS DE SISTEMA

Comprobar si el software cumple con los requisitos especificados.

- Tipos de pruebas
  - **Prueba de recuperación**: se fuerza el fallo del software y que la recuperación se realice correctamente.
  - **Prueba de seguridad**: se comprueba que el sistema esté protegido frente a acciones ilegales.
  - **Prueba de resistencia (Stress)**. Se realizan acciones que requieran una gran cantidad de recursos para comprobar la respuesta de la aplicación.
  - **Pruebas de rendimiento**. Similares a las anteriores pero para sistemas que tienen que cumplir unas especificaciones de rendimiento fijas.
  - **Pruebas de despliegue**. Se realizan para aquellas aplicaciones multiplataforma.

ESTRATEGIA DE PRUEBAS

PRUEBAS DE ACEPTACIÓN O VALIDACIÓN

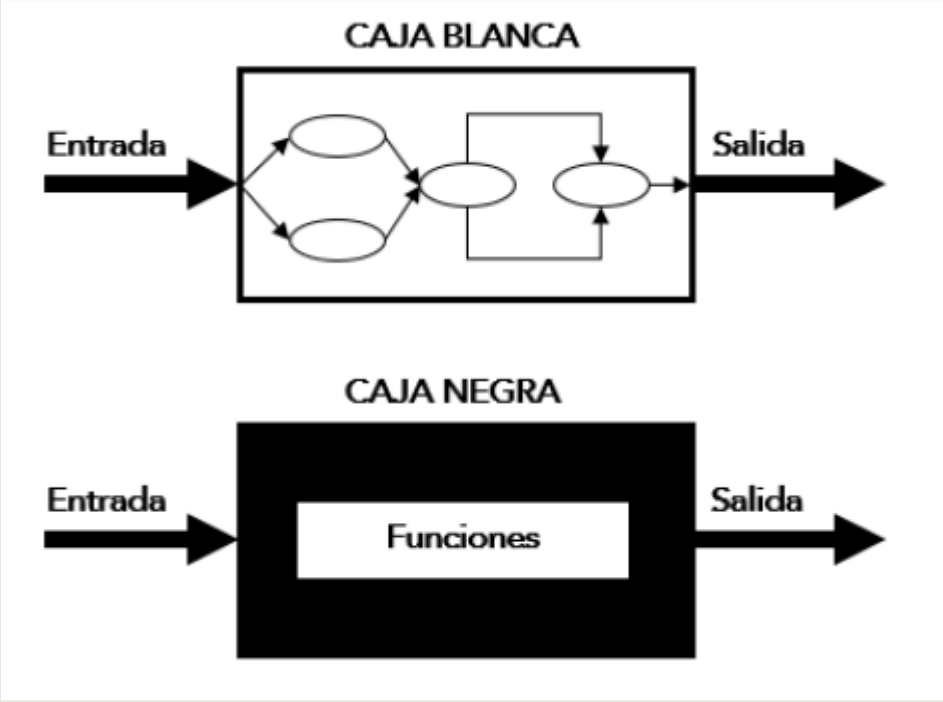
Conseguiremos la prueba de validación cuando el programa funcione de acuerdo con las expectativas expuestas por el cliente y cuando, además, cumpla con lo indicado en el documento de especificación de requisitos del software o ERS.

- Tipos de pruebas
  - **Prueba de recuperación**: se fuerza el fallo del software y que la recuperación se realice correctamente.
  - **Prueba de seguridad**: se comprueba que el sistema esté protegido frente a acciones ilegales.
  - **Prueba de resistencia (Stress)**. Se realizan acciones que requieran una gran cantidad de recursos para comprobar la respuesta de la aplicación.
  - **Pruebas de rendimiento**. Similares a las anteriores pero para sistemas que tienen que cumplir unas especificaciones de rendimiento fijas.
  - **Pruebas de despliegue**. Se realizan para aquellas aplicaciones multiplataforma.

TIPOS DE PRUEBA

TIPOS DE PRUEBAS

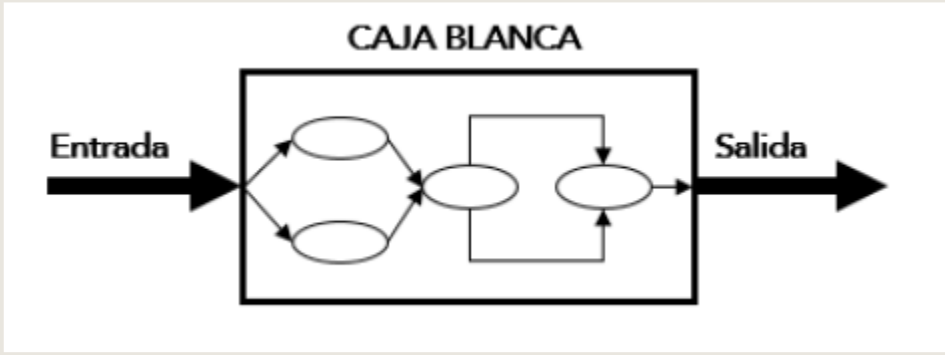
- Tipos de pruebas
  - **Caja blanca o funcionales**. Valida la estructura interna del sistema
  - **Caja negra o estructurales**. Valida los requisitos funcionales sin observar el funcionamiento interno del programa
- No son pruebas excluyentes y podemos combinarlas para descubrir distintos tipos de error



TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA

- Objetivos
  - Asegurar que se ejecutan por lo menos una vez todos los caminos de cada módulo.
  - Todas las sentencias sean ejecutadas al menos una vez.
  - Todas las decisiones lógicas se ejecuten al menos una vez en parte verdadera y otra en la falsa.
  - Todos los bucles sean ejecutados en sus límites.
  - Asegurar que se usen todas las estructuras de datos internas de forma válida
- Características
  - Se examina el código fuente y su ejecución.
  - Se comprueban los flujos de ejecución dentro de cada
  - También pueden comprobarse los flujos entre unidades durante la integración.
- Principales técnicas:
  - Cobertura lógica
  - Prueba de bucles



TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Nos preguntamos cómo crear los casos de prueba para nuestras aplicaciones

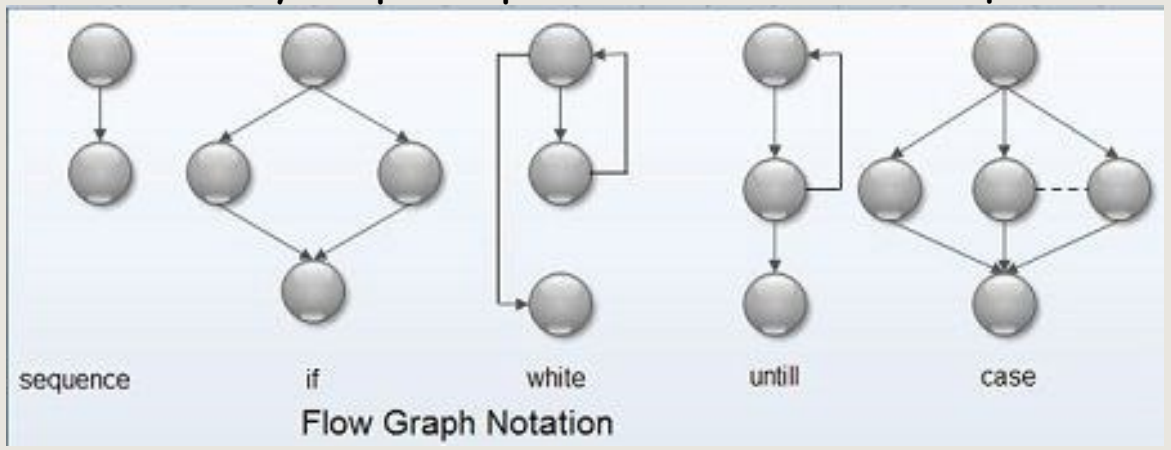
- Esta técnica permite al desarrollador obtener la medida de la complejidad de nuestro sistema.
- Puede usar esta medida para la definición de un conjunto de caminos de ejecución que aseguren que se prueben todas las rutas del código.
- Para obtener esta medida de complejidad utilizaremos la técnica de representación de grafo de flujo

TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Elementos constituyentes

- Elementos gráficos
  - Cada círculo representa una o más sentencias
  - Se numera cada símbolo y los finales
  - Cada círculo es un nodo
  - Cada flecha es una arista
  - Definiremos regiones: áreas que estarán delimitadas por aristas y nodos. Cabe destacar que el área exterior del nodo es otra región más.
  - El nodo predicado contendrá una condición y su principal característica es que salen dos o más aristas de él.



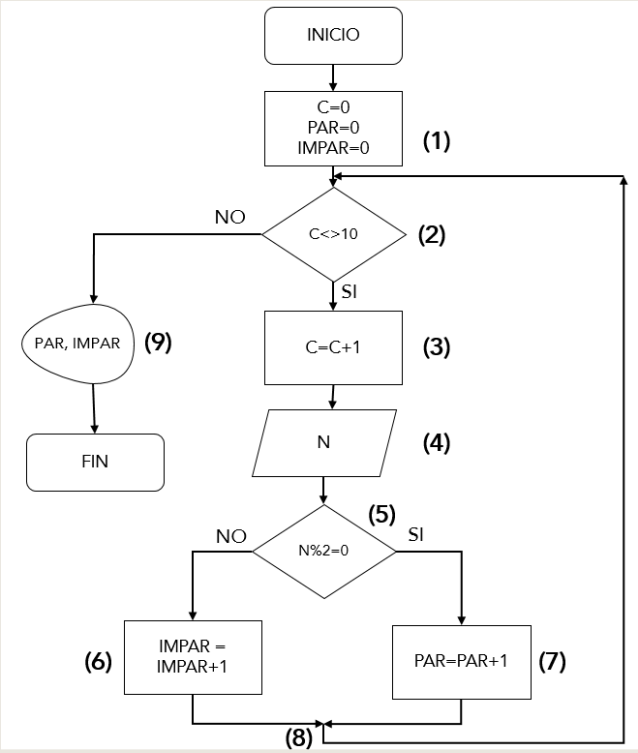
Flow Graph Notation

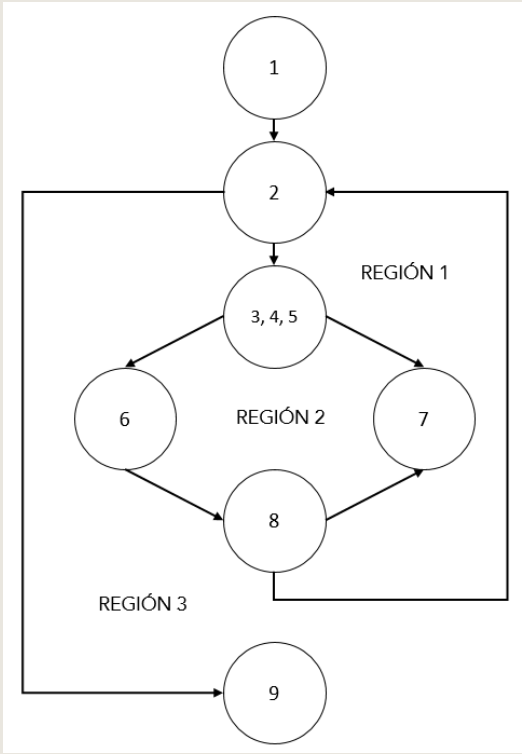
TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Cómo crear un gráfico

- Partiendo del diagrama de flujo o pseudocódigo recorreremos de arriba abajo e iremos creando el diagrama.





TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Cómo crear un gráfico

Inicio

Abrir fichero alumnos

Leer Alumnos (Curso, Nombre, Sexo, Nota)

Mientras Haya registros **Hacer**

NH = 0, NM = 0

Visualizar Curso

Mientras Haya registros **y** Mismo curso **Hacer**

Si Sexo = "H" **Entonces**

NH = NH+1

Si no

NM = NM+1

Fin si

Leer Alumnos (Curso, Nombre, Sexo, Nota)

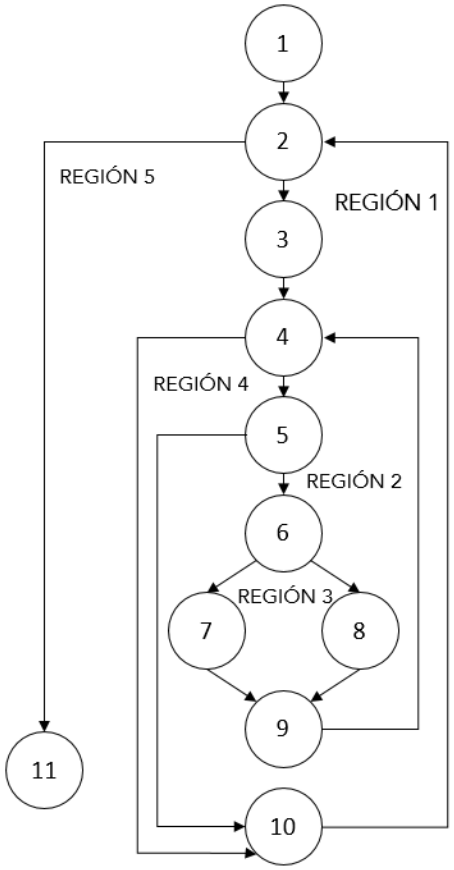
Fin Mientras

Visualizar NH, NM

Fin Mientras

Abrir Fichero Alumnos

Fin



TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Elementos constituyentes

- Cálculo de la complejidad ciclomática
  - Nos establecerá el número de casos de prueba que deberán ejecutarse para que las sentencias sean ejecutadas al menos una vez
  - El valor V(G) nos va a dar el número de caminos independientes del conjunto básico de un programa.
  - Forma de cálculo
    - V(G) = Número de regiones del grafo
    - V(G) = Aristas – Nodos + 2
    - V(G) = Nodos predicado + 1

Complejidad ciclomática	Evaluación del riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado
Entre 21 y 50	Programas o métodos complejos, alto riesgo
Mayor que 50	Programas o métodos no testeables, muy alto riesgo

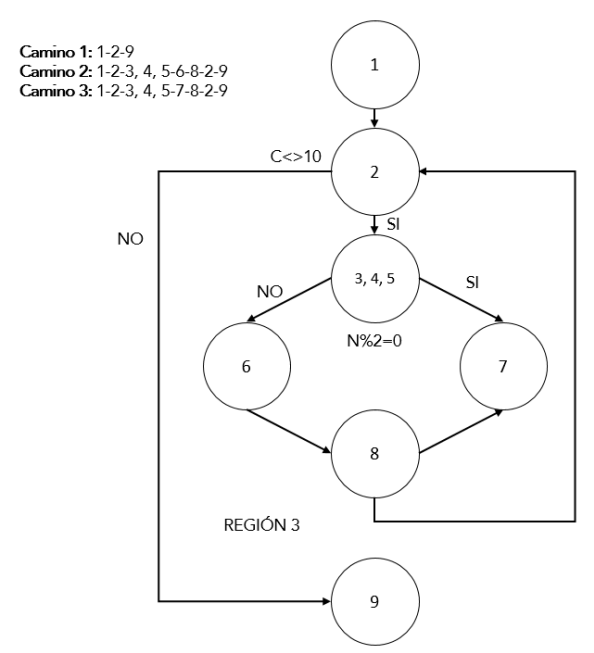
TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Elementos constituyentes

- Desarrollo de los casos de prueba
  - Construir los casos de prueba que fuerzan la ejecución de cada camino. Para comprobar cada camino escogeremos los casos de prueba de tal forma que las condiciones de los nodos predicado estén establecidas adecuadamente.

- V(G) = 3 → Tres caminos



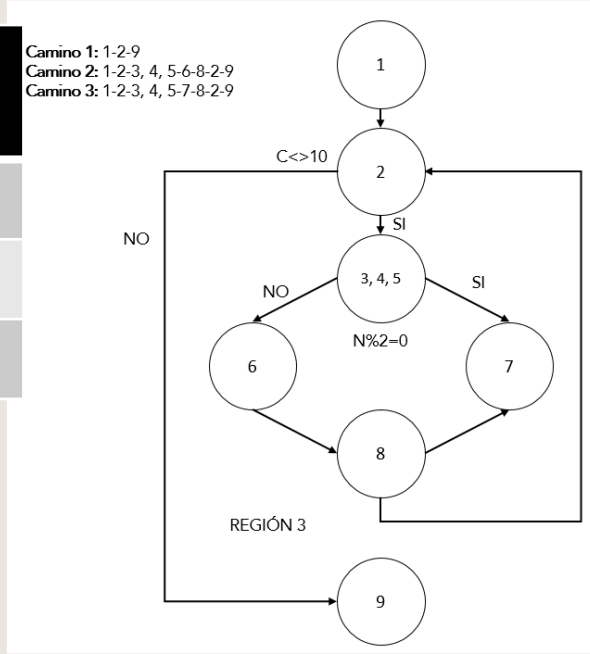
TIPOS DE PRUEBA

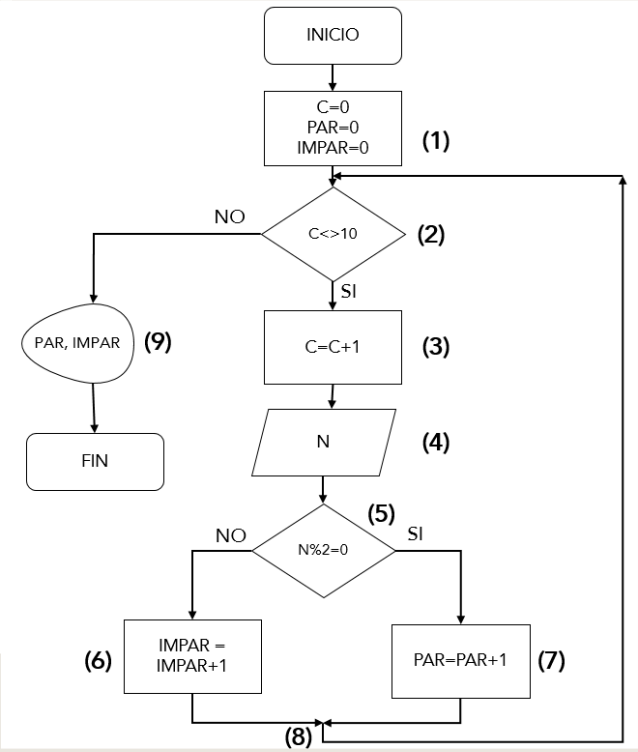
TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DEL CAMINO BÁSICO

Elementos constituyentes

- Desarrollo de los casos de prueba

Camino	Caso de prueba	Resultado esperado
Camino 1	C = 10	Par = Impar = 0
Camino 2	C = 0 N = 1	Par = 0 Impar = 1
Camino 3	C = 0 N = 2	Par = 1 Impar = 0





TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – PRUEBA DE BUCLES

Nos centramos exclusivamente en los bucles

- Para bucles simples
  - Saltar el bucle
  - Realizar una iteración
  - Realizar dos iteraciones
  - Realizar un número menor de iteraciones que el máximo pero mayor que dos
  - Realizar las iteraciones n-1, n y n+1 para un bucle de n vueltas
- Para bucles anidados
  - Comenzar por el bucle más interno y el número mínimo para el resto.
  - Realizar las pruebas para bucles simples para ese bucle.
  - Repetir el proceso saliendo un nivel de anidamiento
  - Continuar hasta probar todos

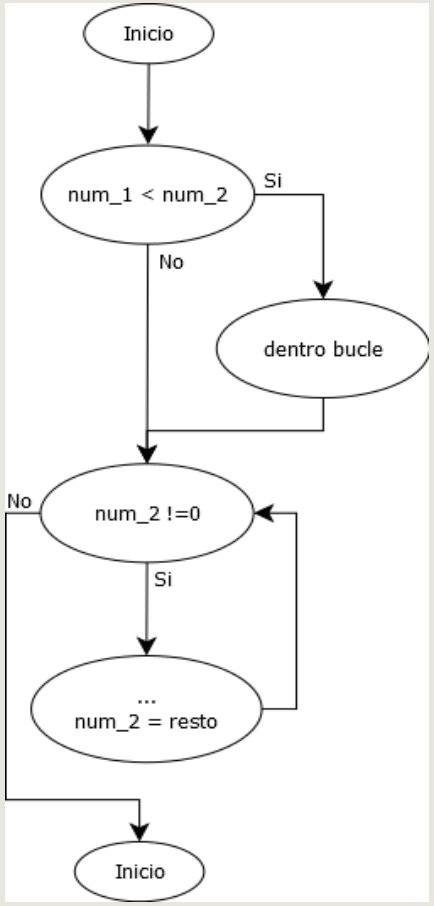
TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – EJERCICIOS

Para el siguiente algoritmo crear los casos de prueba

- Algoritmo de Euclides para el Mcd

```
num_1 = 0
num_2 = 0
resto = 0
print("Introduce dos número")
num_1 = int(input("Num_1:"))
num_2 = int(input("Num_2:"))
if num_1 < num_2:
    aux = num_1
    num_1 = num_2
    num_2 = aux

while num_2 != 0 :
    resto = num_1 % num_2
    num_1 = num_2
    num_2 = resto

print(num_1)
```



¿Cuántos casos de prueba según el camino básico? 4

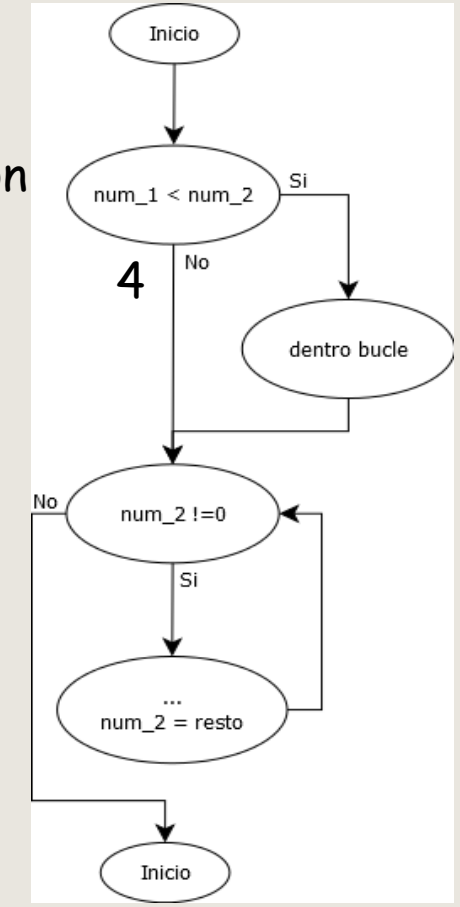
TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – EJERCICIOS

Para el siguiente algoritmo crear los casos de prueba: con IA

- Primero, refactoricemos el código para poner la lógica dentro de una función

```
def calcular_mcd(num_1, num_2):
    """Calcula el Máximo Común Divisor (MCD) de dos números."""
    if num_1 < num_2:
        num_1, num_2 = num_2, num_1

    while num_2 != 0:
        num_1, num_2 = num_2, num_1 % num_2

    return num_1
```



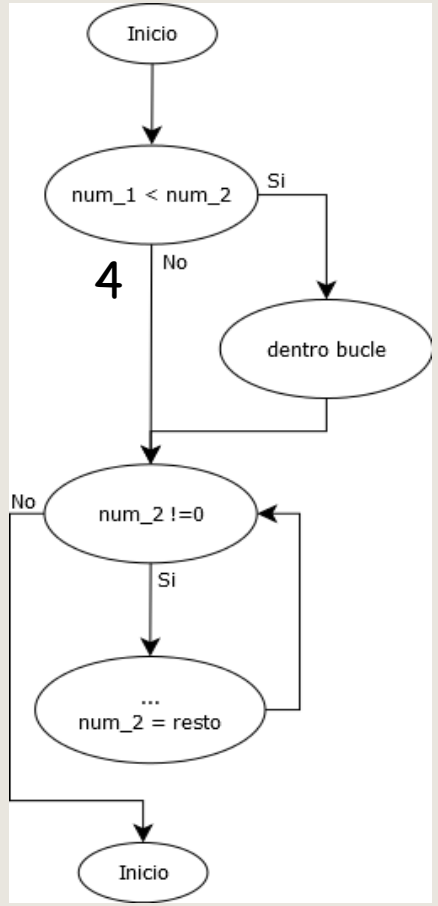
TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – EJERCICIOS

Para el siguiente algoritmo crear los casos de prueba: con IA

- Ahora ya podemos crear los casos de prueba:

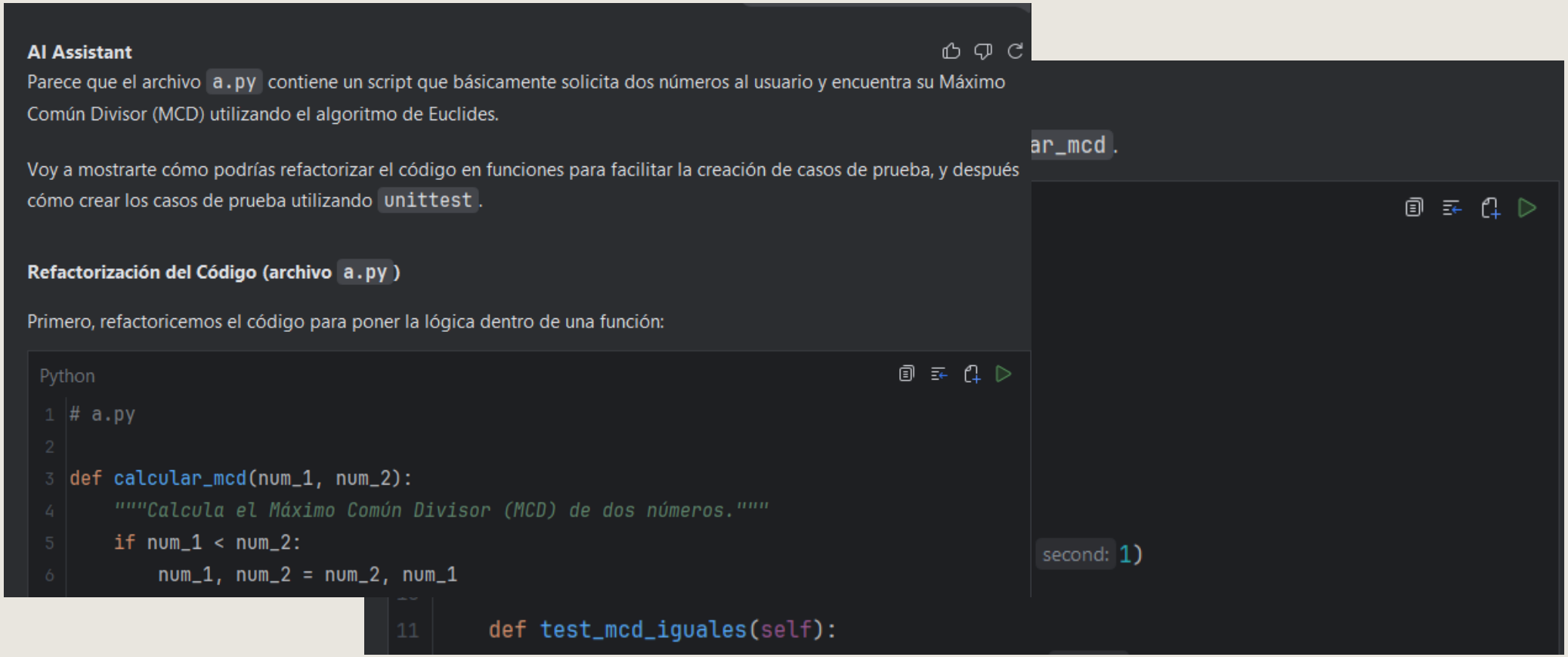
```
import unittest
from a import calcular_mcd
class TestCalcularMCD(unittest.TestCase):
    def test_mcd_primos(self):
        self.assertEqual(calcular_mcd(13, 7), 1)
    def test_mcd_iguales(self):
        self.assertEqual(calcular_mcd(10, 10), 10)
    def test_mcd_general(self):
        self.assertEqual(calcular_mcd(54, 24), 6)
        self.assertEqual(calcular_mcd(48, 18), 6)
    def test_mcd_uno_cero(self):
        self.assertEqual(calcular_mcd(0, 5), 5)
        self.assertEqual(calcular_mcd(5, 0), 5)
    def test_mcd_ambos_cero(self):
        self.assertEqual(calcular_mcd(0, 0), 0)

if __name__ == '__main__':
    unittest.main()
```



TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA - EJERCICIOS

Para el siguiente algoritmo crear los casos de prueba: con IA



TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – EJERCICIOS

Para el siguiente algoritmo crear los casos de prueba

```
while num > 10 :
    print(num)
    num -= 1
```

¿Cuántos casos de prueba según el camino básico? 2

¿Cuántos casos de prueba según bucles? 6

¿Cuáles?

num = 0
num = 11
num = 12
num = 15
num = 9
num = 10

TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA BLANCA – EJERCICIOS

Para los algoritmos de la unidad anterior siguientes crear los casos de prueba

- Dados 10 números enteros introducidos por teclado, visualizar la suma de los pares , cuántos pares existen y cuál es la media aritmética de los impares (002.py).
- Crear y depurar una algoritmo que determine si una subcadena está dentro de una cadena, ambas pedidas al usuario (004.py).
- Crear el juego piedra-papel-tijera (008.py).

TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA NEGRA

- Objetivos
  - Demostrar que las funciones del software son operativas
  - Revisar los requisitos funcionales del programa
- Características
  - Se estudia el sistema desde fuera.
  - Se trabaja sobre la interfaz.
  - No se tienen en cuenta los detalles internos de funcionamiento.
  - Se proporcionan entradas y se estudian las salidas.
  - Principales técnicas:
    - Particiones de equivalencia.
    - Valores límite.

TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA NEGRA – PRUEBA DE PARTICIÓN

- Divide los valores de los campos de entrada de un programa en clases de equivalencia.
- Examinaremos cada condición de entrada para poder identificar estas clases de equivalencia y lo dividiremos en dos o más grupos. Se podrá definir dos tipos de clases de equivalencia:
  - Clase válidas: valores de entrada válidos.
  - Clase no válidas: valores de entrada no válidos.
- Directrices para crear las clases
  - Si requiere un valor específico, define una clase válida y dos no válidas (inferior y superior).
  - Si una condición de entrada requiere un rango, se define una clase de equivalencia válida (dentro del rango) y dos no válidas (extremos del rango).
  - Si especifica un miembro de un conjunto, define una válida y una no válida.
  - Si es lógica, define una clase válida y una no válida.
- Procedimiento
  - Asignar un número único a cada clase de equivalencia
  - Crear el número mínimo de casos de prueba que incluyan todas las clases válidas
  - Crear un caso de prueba por cada clase no válida

TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA NEGRA – ANÁLISIS LOS VALORES LÍMITE

- Se basa en la hipótesis de que suelen ocurrir más errores en los valores extremos de los campos de entrada
- Reglas
  - Si una condición de entrada especifica un rango de valores, deberemos concretar casos de prueba para los límites del rango y para los valores justo por encima y por debajo. Ejemplo: para un rango de valores enteros que estén comprendidos entre 5 y 15, tenemos que escribir casos de prueba para 5, 15, 4 y 16.
  - Si especifica un rango de valores, similar al anterior.
  - Para la condición de salida aplicaremos la regla 1.
  - Usar también para la condición de salida la regla 2. Tanto en esta regla como en la anterior no se generarán valores que estén fuera del rango.
  - Si la estructura interna posee límites prestablecidos nos aseguraremos de diseñar casos de prueba que ejerciten la estructura de datos en sus límites, primer y último elemento.

TIPOS DE PRUEBA

TIPOS DE PRUEBAS: CAJA NEGRA – PRUEBA DE INTERFACES

- Determinar si una interfaz de usuario funciona correctamente
- Reglas
  - Usar datos reales.
  - Mejor si lo hace un usuario en vez del programador

AUTOMATIZACIÓN DE LAS PRUEBAS

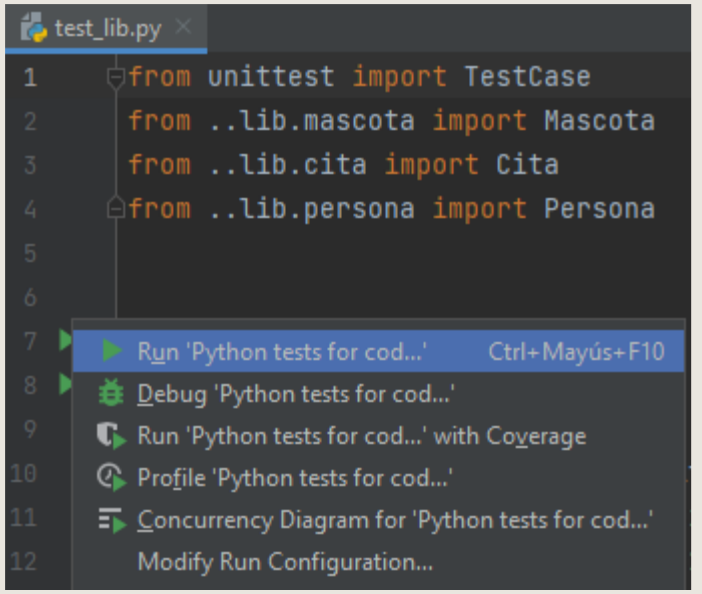
INTRODUCCIÓN

- El desarrollo de pruebas se hace bajo el módulo unittest
- Para hacer pruebas crearemos un fichero con una clase nuestra que herede de TestCase
- Cada método de esa clase deberá comenzar por test_xxxx
- En cada método haremos
  - Preparar el entorno (opcional).
  - Prepara el resultado esperado.
  - Llama el código bajo prueba.
  - Asegúrate que el resultado real coincida con el resultado esperado.
- Una vez definida la clase con todos los métodos se ejecutará la clase
- Automáticamente se llamarán a todos los métodos
- Las pruebas se hacen dentro del método con assert
- Hay gran cantidad de pruebas a realizar

AUTOMATIZACIÓN DE LAS PRUEBAS

EJEMPLO

```
import unittest
class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)
```



AUTOMATIZACIÓN DE LAS PRUEBAS

LISTADO DE ASSERTS

Método	Comprueba
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

AUTOMATIZACIÓN DE LAS PRUEBAS

LISTADO DE ASSERTS

Método	Comprueba
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises *exc*
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises *exc* and the message matches regex *r*
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises *warn*
assertWarnsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) raises *warn* and the message matches regex *r*
assertLogs(logger, level)	The with block logs on *logger* with minimum *level*
assertNoLogs(logger, level)	The with block does not log on*logger* with minimum *level*

AUTOMATIZACIÓN DE LAS PRUEBAS

LISTADO DE ASSERTS

Método	Comprueba
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	not r.search(s)
assertCountEqual(a, b)	*a* and *b* have the same elements in the same number, regardless of their order.

AUTOMATIZACIÓN DE LAS PRUEBAS

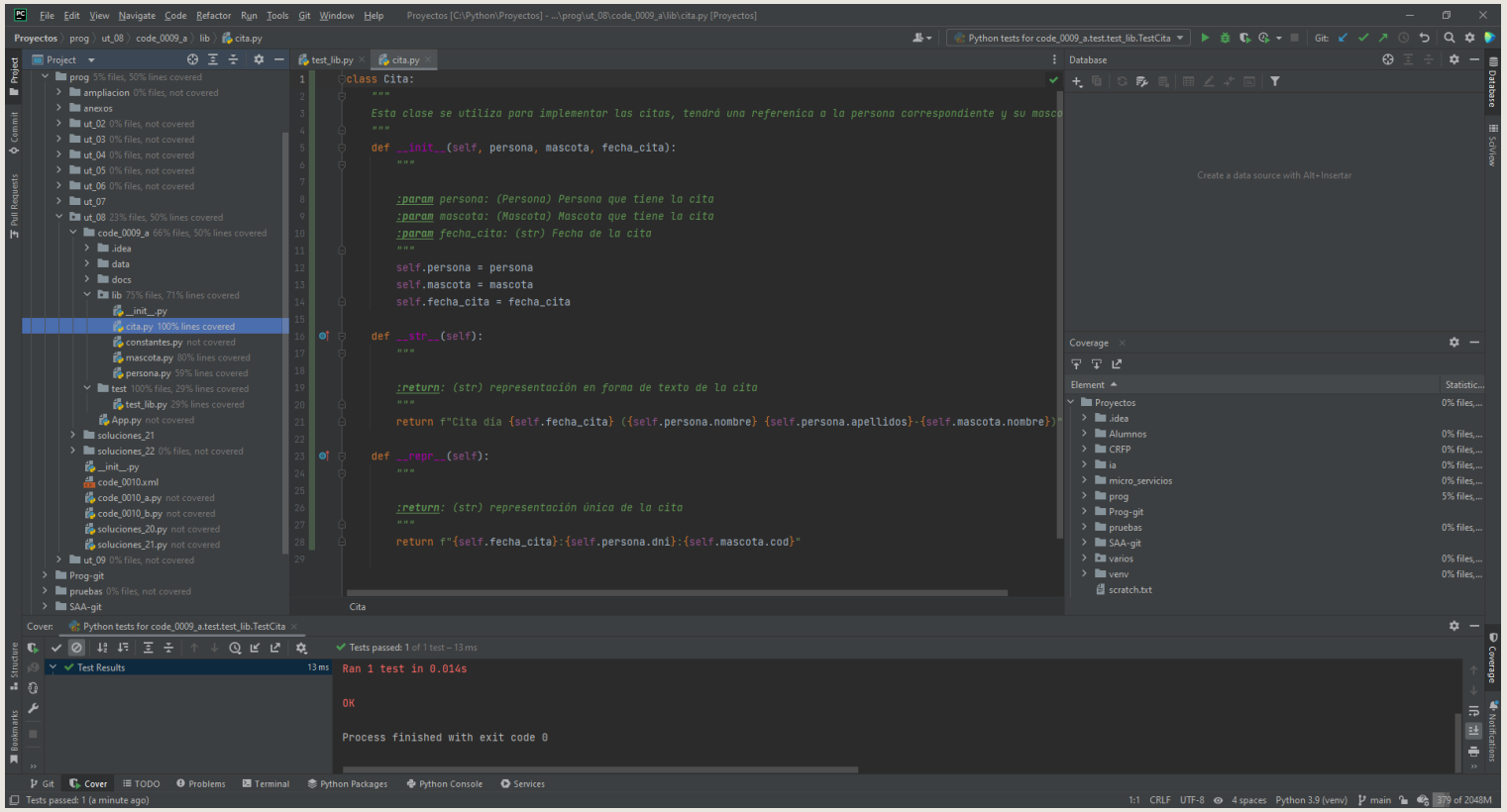
LISTADO DE ASSERTS

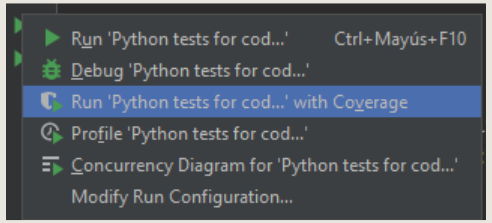
Method	Used to compare
assertMultiLineEqual(a, b)	strings
assertSequenceEqual(a, b)	sequences
assertListEqual(a, b)	lists
assertTupleEqual(a, b)	tuples
assertSetEqual(a, b)	sets or frozensets
assertDictEqual(a, b)	dicts

AUTOMATIZACIÓN DE LAS PRUEBAS

COBERTURA DEL CÓDIGO

- La cobertura del código nos indicará el porcentaje del código fuente que se prueba
- Lo ideal es que se pruebe el 100%
- Nos indicará qué partes se prueban y cuáles no





AUTOMATIZACIÓN DE LAS PRUEBAS

EJEMPLO

- Dados 10 números enteros, devolver la suma de los pares e impares, cuántos pares e impares existen (f_009.py).

```
def calcular(numeros):
    numero_actual = suma_pares = total_pares = total_impares = suma_impares = 0
    while numero_actual < len(numeros):
        numero = numeros[numero_actual]
        if numero % 2 == 0:  # par
            suma_pares += numero
            total_pares += 1
        else:
            suma_impares += numero
            total_impares += 1
        numero_actual += 1

    return suma_pares, total_pares, suma_impares, total_impares
```

AUTOMATIZACIÓN DE LAS PRUEBAS

EJEMPLO

- Dados 10 números enteros, devolver la suma de los pares e impares, cuántos pares e impares existen (f_009.py).

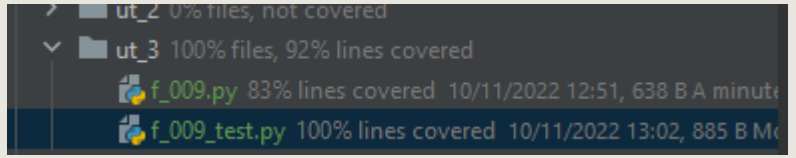
```
from unittest import TestCase
from f_009 import calcular
class TestCalculo(TestCase):
    def test_calculara(self):
        self.assertEqual(calcular([1, 1, 1, 1, 1]), (0, 0, 5, 5))
        self.assertEqual(calcular([2, 2, 2, 2, 2]), (10, 5, 0, 0))
        #  En este momento ya está el 100% del código cubierto
        self.assertEqual(calcular([2, 1, 1, 1, 1]), (2, 1, 4, 4))
        self.assertEqual(calcular([2, 2, 2, 2, 1]), (8, 4, 1, 1))
        self.assertEqual(calcular([1, 1, 1, 1, 2]), (2, 1, 4, 4))
        self.assertEqual(calcular([1, 2, 2, 2, 2]), (8, 4, 1, 1))
        self.assertEqual(calcular([2, 2, 1, 1, 1]), (4, 2, 3, 3))
        self.assertEqual(calcular([2, 1, 1]), (2, 1, 2, 2))
        self.assertEqual(calcular([2]), (2, 1, 0, 0))
        self.assertEqual(calcular([1]), (0, 0, 1, 1))
        self.assertEqual(calcular([]), (0, 0, 0, 0))
```

AUTOMATIZACIÓN DE LAS PRUEBAS

EJEMPLO

- Dados 10 números enteros, devolver la suma de los pares e impares, cuántos pares e impares existen (f_009.py).

```python
def calcular(numeros):
    numero_actual = suma_pares = total_pares = total_impares = suma_impares = 0
    while numero_actual < len(numeros):
        numero = numeros[numero_actual]
        if numero % 2 == 0:  # par
            suma_pares += numero
            total_pares += 1
        else:
            suma_impares += numero
            total_impares += 1
        numero_actual += 1

    return suma_pares, total_pares, suma_impares, total_impares
```



AUTOMATIZACIÓN DE LAS PRUEBAS

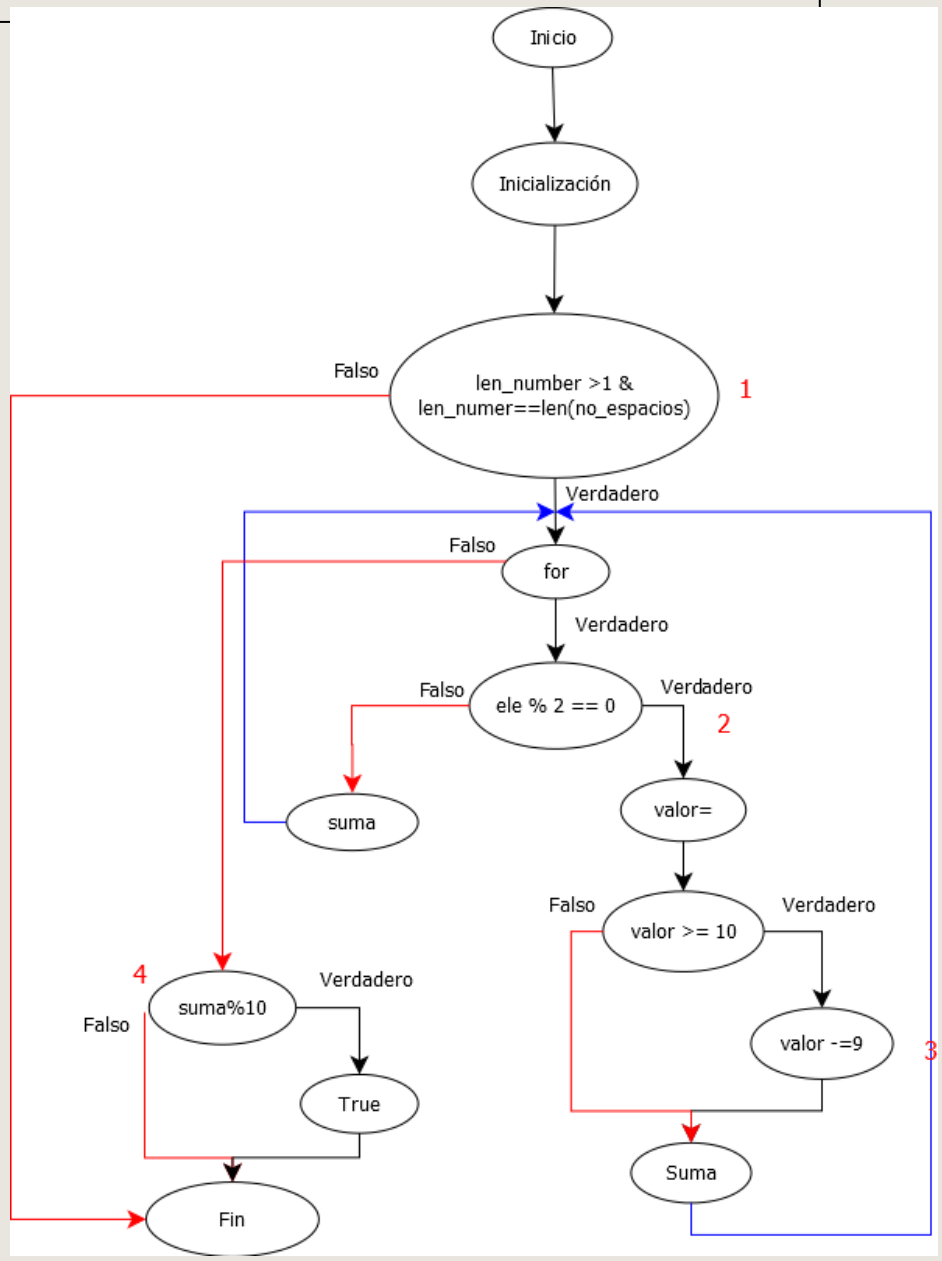
EJERCICIO (UT_2/006.PY)

```
def valid(card_num):
    number = "".join([s for s in card_num if s.isdigit()])
    number_no_spaces = card_num.replace(" ", '')
    ret = False
    suma = 0
    len_number = len(number)
    if len_number > 1 and len_number == len(number_no_spaces):
        for ele in range(1, len_number + 1, 1):
            if ele % 2 == 0:
                valor = int(number[len_number - ele]) * 2
                if valor >= 10: valor -= 9
                suma += valor
            else:
                suma += int(number[len_number - ele])

        if suma % 10 == 0:
            ret = True
    return ret
```

AUTOMATIZACIÓN DE LAS PRUEBAS

EJERCICIO: CAMINO BÁSICO

```
def valid(card_num):
    number = "".join([s for s in card_num if s.isdigit()])
    number_no_spaces = card_num.replace(" ", '')
    ret = False
    suma = 0
    len_number = len(number)
    if len_number > 1 and len_number == len(number_no_spaces):
        for ele in range(1, len_number + 1, 1):
            if ele % 2 == 0:
                valor = int(number[len_number - ele]) * 2
                if valor >= 10: valor -= 9
                suma += valor
            else:
                suma += int(number[len_number - ele])

        if suma % 10 == 0:
            ret = True
    return ret
```



AUTOMATIZACIÓN DE LAS PRUEBAS

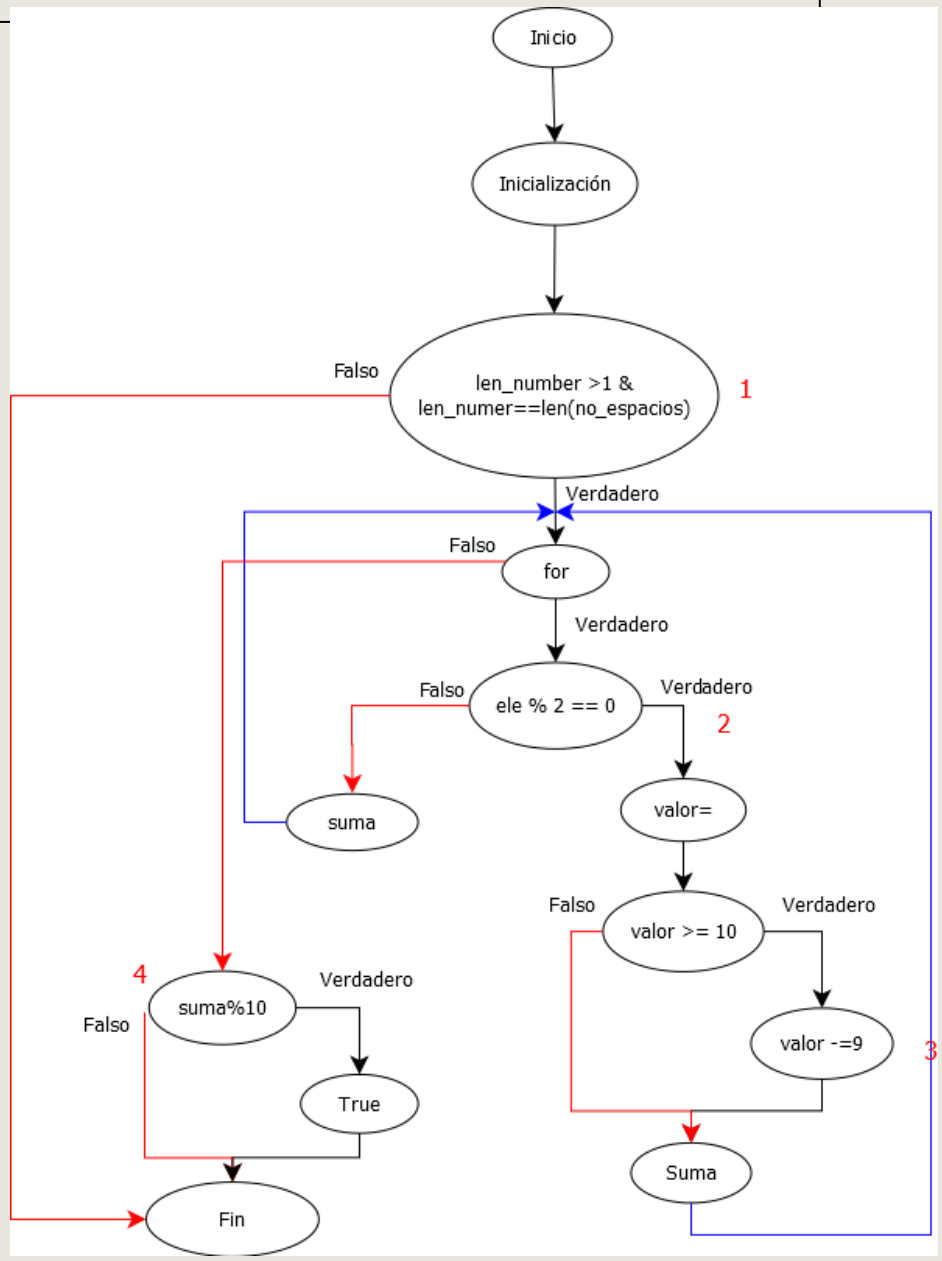
EJERCICIO: CAMINO BÁSICO

Prueba número 1:
- len_number>1 && == → "93" F
- len_number>1 && != → "9b3" F
- len_number<=1 && == → "9" F
- len_number<=1 && != → "b" F

Prueba número 2:
- ele%2 True → "88" F
- ele%2 False→ "091" T

Prueba número 3:
- >=10 → "77" F
- < 10 → "34" T

Prueba número 4:
- %10 True → "055 444 285" T
- %10 False → "055 444 286" F



AUTOMATIZACIÓN DE LAS PRUEBAS

EJERCICIO

Para los algoritmos de la unidad anterior siguientes crear los casos de prueba

- Crear y depurar una algoritmo que determine si una subcadena está dentro de una cadena, ambas pedidas al usuario (004.py).
- Dado un número, determine si es válido o no según la fórmula de Luhn (006.py).
- Crear las tablas de multiplicar del 1 al 10, pero pidiendo al usuario cuál es la última a visualizar y una que haya que saltar (007.py).
- Crear el juego piedra-papel-tijera (008.py).

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

FACTORES

Los factores que afectan a la calidad del software se centran en tres aspectos importantes de un producto software: sus características operativas, su capacidad de soportar los cambios (revisión del producto) y su adaptabilidad a nuevos entornos

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

FACTORES DE CALIDAD DE MCCALL

- **Corrección**. La corrección es la propiedad del software de funcionar correctamente.
- **Robustez**. La relacionamos con la capacidad de un sistema de software de estar preparado para un funcionamiento correcto ante las excepciones que pudieran producirse.
- **Eficiencia**. Cantidad de recursos de computadora y de código requeridos por un programa para llevar a cabo sus funciones.
- **Integridad**. Grado en que puede controlarse el acceso al software o a los datos, por personal no autorizado
- **Facilidad de uso**. Esfuerzo requerido para aprender un programa, trabajar con él, preparar su entrada e interpretar su salida
- **Mantenibilidad**. Esfuerzo requerido para localizar y arreglar un error en un programa

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

FACTORES DE CALIDAD DE MCCALL

- **Flexibilidad**. Esfuerzo requerido para modificar un programa operativo.
- **Facilidad de prueba**. Esfuerzo requerido para probar un programa de forma que se asegure que realiza la función requerida
- **Portabilidad**. Esfuerzo requerido para transferir el programa desde un hardware y/o entorno de sistemas de software a otro
- **Reusabilidad**. Grado en que un programa (o partes de un programa) puede reusarse en otras aplicaciones
- **Interoperatividad**. Esfuerzo requerido para acoplar un sistema a otro

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

MÉTRICAS

Es difícil (y en algunos casos imposible) desarrollar medidas directas de los anteriores factores de calidad. Por tanto, se define un conjunto de métricas usadas para evaluar los factores de calidad

El esquema de puntuación propuesto por McCall es una escala del 0 (bajo) al 10 (alto), sobre las siguientes métricas. Se conseguirán los datos a través de encuestas.

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

MÉTRICAS DE CALIDAD DE MCCALL

- **Facilidad de auditoría**. La facilidad con que se puede comprobar la conformidad con los estándares
- **Exactitud**. La precisión de los cálculos y del control.
- **Normalización de las comunicaciones**. El grado en que se usa el ancho de banda, los protocolos y las interfaces estándar.
- **Completitud**. El grado en que se ha conseguido la total implantación de las funciones requeridas.
- **Concisión**. Lo compacto que es el programa en términos de líneas de código.
- **Consistencia**. El uso de un diseño uniforme y de técnicas de documentación a lo largo de todo el programa.
- **Estandarización en los datos**. El uso de estructuras de datos y de tipos estándar a lo largo de todo el programa.

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

MÉTRICAS DE CALIDAD DE MCCALL

- **Tolerancia de errores**. El daño que se produce cuando el programa detecta una situación errónea.
- **Eficiencia en la ejecución**. El rendimiento en tiempo de ejecución de un programa
- Facilidad de expansión. El grado en que se puede ampliar el diseño arquitectónico, de datos o procedimental
- **Generalidad**. La amplitud de aplicación potencial de los componentes del programa
- **Independencia del hardware**. El grado en que el software es independiente del hardware sobre el que opera
- **Instrumentación**. El grado en que el programa muestra su propio funcionamiento e identifica errores que aparecen
- **Modularidad**. La independencia funcional de los componentes del
- programa
- **Facilidad de operación**. La facilidad de utilización de un programa

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

CONTROL DE CALIDAD

Control de calidad es la medición de la calidad de un producto mediante pruebas

MÉTRICAS DE CALIDAD DE MCCALL

- **Seguridad**. La disponibilidad de mecanismos que controlen o protejan los programas o los datos
- **Autodocumentación**. El grado en que el código fuente proporciona documentación significativa
- **Simplicidad**. El grado en que un programa puede ser entendido sin dificultad
- **Independencia del sistema de software**. El grado en que el programa es independiente de características no estándar del lenguaje de programación, de las características del sistema operativo y de otras restricciones del entorno
- **Facilidad de traza**. La posibilidad de seguir la pista a la representación del diseño o de los componentes reales del programa hacia atrás, hacia los requisitos
- **Formación**. El grado en que el software ayuda a permitir que nuevos usuarios empleen el sistema

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

RELACIÓN MÉTRICAS - FACTORES

MEDIDA DE LOS FACTORES DE CALIDAD

Criterio \ Factor de calidad	A. Corrección	B. Fiabilidad	C. Eficiencia	D. Seguridad (Integridad)	E. Usabilidad	F. Facilidad de Mantenimiento	G. Flexibilidad	H. Facilidad de Pruebas	I. Portabilidad	J. Reusabilidad	K. Interoperatividad
01. Facilidad de auditoría (FA)				X				X			
02. Exactitud (EX)		X									
03. Estandarización de comunicaciones (EC)											X
04. Compleción (CM)	X										
05. Complejidad (CX)		X					X	X			
06. Concisión (CN)			X			X	X				
07. Consistencia (CS)	X	X				X	X				
08. Estandarización de datos (ED)											X
09. Tolerancia al error (TE)		X									
10. Eficiencia de ejecución (EE)			X								
11. Capacidad de expansión (CE)							X				
12. Generalidad (GE)							X		X	X	X
13. Independencia del hardware (IH)									X	X	
14. Instrumentación (IN)				X		X		X			
15. Modularidad (MD)		X				X	X	X	X	X	X
16. Operatividad (OP)			X		X						
17. Seguridad (SG)				X							
18. Autodocumentación (AD)						X	X	X	X	X	
19. Simplicidad (SM)		X				X	X	X	X		
20. Independencia del sistema software (IS)										X	
21. Trazabilidad (TZ)	X										
22. Formación (Entrenamiento) (FM)					X						

Relación entre Criterios y Factores de calidad según McCall

CALIDAD DEL SOFTWARE (AMPLIACIÓN)

RELACIÓN MÉTRICAS - FACTORES

Los factores C se les debe dar valor, un uso muy normal es que sumen 1 en cada expresión

Las relaciones entre los factores de calidad y los criterios son las siguientes:

A. Corrección = (C1 * CM) + (C2 * CS) + (C3 * TZ)

B. Fiabilidad = (C1 * EX) + (C2 * CX) + (C3 * CS) + (C4 * TE) + (C5 * MD) + (C6 * SM)

C. Eficiencia = (C1 * CN) + (C2 * EE) + (C3 * OP)

D. Seguridad (Integridad) = (C1 * FA) + (C2 * IN) + (C3 * SG)

E. Usabilidad (Facilidad de uso) = (C1 * OP) + (C2 * FM)

F. Facilidad de mantenimiento = (C1 * CN) + (C2 * CS) + (C3 * IN) + (C4 * MD) + (C5 * AD) + (C6 * SM)

G. Flexibilidad =
(C1 * CX) + (C2 * CN) + (C3 * CS) + (C4 * CE) + (C5 * GE) + (C6 * MD) + (C7 * AD) + (C8 * SM)

H. Facilidad de Pruebas = (C1 * FA) + (C2 * CX) + (C3 * IN) + (C4 * MD) + (C5 * AD) + (C6 * SM)

I. Portabilidad = (C1 * GE) + (C2 * IH) + (C3 * MD) + (C4 * AD) + (C5 * SM)

J. Reusabilidad = (C1 * GE) + (C2 * IH) + (C3 * MD) + (C4 * AD) + (C5 * IS)

K. Interoperatividad = (C1 * EC) + (C2 * ED) + (C3 * GE) + (C4 * MD)

FIN U.T.3

Una prueba de software es un proceso por medio del cual se evalúa la funcionalidad de un software y se intenta identificar posibles errores.

Su propósito principal es asegurar que la aplicación desarrollada cumpla con los estándares y se ofrezca al cliente un producto de calidad.