



4. Testing

Curso 2023-2024

Entornos de desarrollo
Desarrollo de Aplicaciones Web

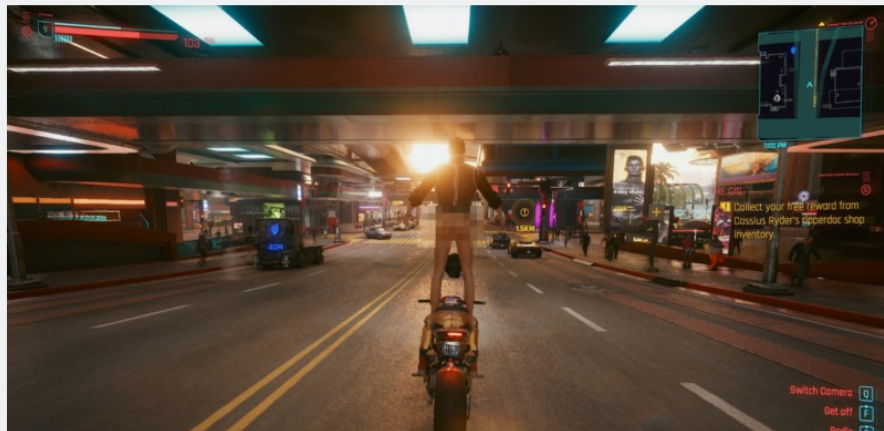
¿Cómo sabéis si un programa que habéis hecho está bien? ¿Al 100%? ¿Os jugaríais dinero?

LOOK THE OTHER WAY —

CDPR admits it “ignored the signals” of *Cyberpunk 2077*’s console issues

Refund requests are dependent on console/retailer policies, CDPR says.

KYLE ORLAND - 12/15/2020, 6:18 PM



4. Testing

Testing

Como ya vimos, uno de los roles que suele haber en un equipo es el tester, o QA engineer, quien se encarga de ***hacer pruebas, testear, verificar y validar que una aplicación software funciona de acuerdo a las especificaciones abordadas.***

Dichos ingenieros emplean estrategias para abordar el software a probar, utilizando métodos de testeo y planificando qué elementos probar, integrar todos los elementos de un sistema...

Spain / Job / Quality Assurance (QA) Engineer

Average Quality Assurance (QA) Engineer Salary in Spain

Pay

Job Details

Skills

Job Listings

How should I pay?

Price a Job

What am I worth?

Find market worth

€32,000 / year

Avg. Base Salary (EUR)



The average salary for a Quality Assurance (QA) Engineer is €32,000

Base Salary ⓘ

€20k - €42k

Bonus

€2k - €3k

Total Pay ⓘ

€20k - €43k

Currency: EUR • Updated: Mon Dec 14 2020 •

Individuals Reporting: 41

Based on 41 salary profiles (last updated Dec 14 2020)

Quality assurance en Canarias, España

427 resultados

Crear alerta



**QA who just
tests as per
documents**

**QA who
understands
product and
use cases**



GENIUS

Realización de pruebas

El testing es **CARO** y **REQUIERE MUCHO TIEMPO**, es prácticamente imposible probar cada detalle de nuestro producto software.

Si nuestro programa no es tan importante como para realizar todas las pruebas posibles, se llega a un punto intermedio en el cual se garantiza que no habrá defectos importantes y la aplicación cumplirá los requisitos mínimos de funcionamiento.

El objetivo de las pruebas es convencer, tanto a los usuarios como a los desarrolladores de que el software desarrollado es lo suficientemente robusto como para su propósito.

El nivel de confianza dependerá de varios factores:

- ¿El rendimiento del software es crítico?
- ¿Hay una base de usuarios suficientemente grande y acostumbrada?
- ¿Cómo es el mercado?

Si un software supera unas pruebas exhaustivas, las probabilidades de que genere errores se atenúan y por tanto, es más fiable.





En el proceso de desarrollo de software, nos vamos a encontrar con un conjunto de actividades, donde **es muy fácil que se produzca un error humano**.

Estos errores humanos pueden ser: una incorrecta especificación de los objetivos, errores producidos durante el proceso de diseño y errores que aparecen en la fase de desarrollo.

Un error, que desde el punto de vista de codificación puede ser relativamente simple de corregir, puede resultar muy difícil y costoso de detectar y puede llegar a tener graves efectos en la organización.

Una de las características típicas del desarrollo de software basado en el ciclo de vida es la realización de controles periódicos, normalmente coincidiendo con los hitos del proyecto o la terminación de documentos. Estos controles pretenden una evaluación de la calidad de los productos generados (especificación de requisitos, documentos de diseño, etc.) para poder detectar posibles defectos cuanto antes

Antes de continuar es necesario conocer las definiciones de los siguientes términos:

- **Prueba (test):** *“Una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto”.*
- **Caso de prueba (test case):** *“Ejecución de un elemento de un sistema especificando el conjunto de entradas, condiciones de ejecución, y resultados esperados desarrollados para un objetivo”.*
- **Defecto (defect, fault, “bug”):** *“Un defecto en el software como, por ejemplo, un proceso, una definición de datos o un paso de variables incorrectos en un programa”.*
- **Fallo (Failure):** *“La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimientos especificados”.*
- **Error (error):**
 - “La diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto”.
 - *“Una acción humana que conduce a un resultado incorrecto”.*



4. Testing

Ejemplo real -> Super Mario 64 (1996)

If you are decompiling the game, you can make this change very easily without using patches:

- Open `/actors/burn_smoke/model.inc.c`
- On line 47, change the reference `"G_IM_FMT_RGBA"` to `"G_IM_FMT_IA"`

At build time, this will compile the texture into IA16 format (correct) instead of RGBA16 (incorrect).

URL: [SuperMario-Error Humano](#)

URL: [Video con el error](#)



- **Depuración:** El proceso de localizar, analizar y corregir los defectos que se sospecha que contiene el software.
- **Verificación:**
 - **Comprobar que el software cumple sus requisitos funcionales y no funcionales (tema1)**
 - *Un conjunto de actividades que aseguran que el software implementa correctamente una función específica.*
 - **¿Are we building the product right?**
- **Validación:**
 - El proceso de evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para **determinar si el software hace lo que el usuario deseaba.**
 - *Un conjunto de actividades que aseguran que el software construido se ajusta a los requisitos del cliente.*
 - **¿Are we building the right product?**

¿Cómo verificar y validar una web? ¿Y un app de móvil? ¿Y un videojuego?

Normalmente los programas simples no van a ser probados con mucho ahínco. Cuando empecemos a programar con sistemas con varios componentes es muy valioso encontrar defectos en los compases iniciales en vez de cuando el sistema ya esté integrado y conectado entre sí. Hay tres fases:

- **Development testing:** Los componentes se testean por los desarrolladores, de forma independiente, y desconectado del resto de componentes. Un componente puede ser un objeto, o un supergrupo de otras entidades.
- **System testing:** Este proceso se ocupa de encontrar errores inesperados en las interacciones entre componentes de un sistema.
- **Acceptance testing:** La fase final, antes de que el sistema se acepte para su uso. Se usan datos proporcionados por los usuarios del sistema. Esta fase puede revelar errores y omisiones en las funcionalidades ya que los datos reales y circunstancias reales difieren mucho de los datos usados durante el testeo.

Los test van guiados por el test plan (plan maestro de pruebas) que han sido especificados y detallados durante la fase de análisis y diseño.

En una metodología ágil, al ser iterativo, el testeo va de la mano del desarrollo (development + system + acceptance)

En la fase de pruebas, se diseñan y preparan los casos de prueba, que se crean con el objetivo de encontrar fallos.

Los casos de prueba deben ser todo lo retorcidos que puedan, y llevar al sistema al límite de forma que trabaje en circunstancias en las que no se use normalmente.

Hay que tener en cuenta que la prueba no debe ser muy sencilla ni muy compleja. Si es muy sencilla, no va a aportar nada y, si es muy compleja, quizá, sea demasiado difícil encontrar el origen de los errores.

Testear es (muy resumidamente) ejecutar casos de prueba uno a uno, pero que un software pase todos los casos de prueba no quiere decir que el programa esté exento de fallos.

Buenas prácticas (Whittaker, [2002](#)):

- **Seleccionar entradas que fuercen a generar todos los mensajes de error.**
- **Usar pruebas que provoquen un desbordamiento de los valores.**
- **Repetir el mismo input varias veces.**
- **Forzar salidas inválidas y erróneas.**
- **Obligar a que los resultados de computación sean muy grandes o muy pequeños.**



Tipos de pruebas

- ✓ **Pruebas funcionales:** buscan que los componentes software diseñados cumplan la función con la que fueron diseñados y desarrollados.
- ✓ **Pruebas no funcionales:** son aquellas pruebas más técnicas que se realizan al sistema. Son de caja negra porque nunca se examina la lógica interna de la aplicación. Ejemplos: pruebas de estrés, pruebas de rendimiento, pruebas de fiabilidad, pruebas de usabilidad...
- ✓ **Pruebas estructurales:** o de caja blanca ya que analizan en algún momento el código fuente para encontrar defectos.
- ✓ **Pruebas de regresión o pruebas repetidas:** Consiste en volver a lanzar previas pruebas para comprobar que algún cambio en algún componente no hay introducido defectos nuevos.

Pruebas de caja negra

Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos. **No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.**

Su principal cometido, va a consistir, en comprobar el correcto funcionamiento de los componentes de la aplicación informática. Para realizar este tipo de pruebas, se deben analizar las entradas y las salidas de cada componente, verificando que el resultado es el esperado. **Solo se van a considerar las entradas y salidas del sistema, sin preocuparnos por la estructura interna del mismo.**

Si por ejemplo, estamos implementando una app para Android que se comunica con una base de datos, en el enfoque de las pruebas funcionales, solo nos interesa verificar que ante una determinada entrada a ese programa el resultado de la ejecución del mismo devuelve como resultado los datos esperados. Este tipo de prueba, no consideraría, en ningún caso, el código desarrollado, ni el algoritmo, ni la eficiencia, ni si hay partes del código innecesarias, etc.



Dentro de las pruebas funcionales, podemos indicar cuatro tipos de pruebas:

- **Particiones equivalentes:** Consiste en **es crear un conjunto de clases de equivalencia para los valores de entrada, separados en casos válidos e inválidos**, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase.
- **Análisis de valores límite:** En este caso, a la hora de implementar un caso de prueba, se van a elegir como valores de entrada, **aquellos que se encuentran en el límite de las clases de equivalencia**.
- **Tablas de decisión:** Las tablas de decisión enfrentan cada una de las condiciones o clases de equivalencia frente a los resultados que se producen, tanto por separado como realizando condiciones entre ellas. Se utilizan en sistemas que trabajan en tiempo real o empotrados.
- **Random testing:** Consiste en generar entradas aleatorias (RGN, datos al azar de una base de datos...) para la aplicación que hay que probar. Se suelen utilizar generadores de prueba, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación. Este tipo de pruebas, se suelen utilizar en aplicaciones que no sean interactivas, ya que es muy difícil generar las secuencias de entrada adecuadas de prueba, para entornos interactivos. (QuickCheck, Randoop)
- **Hipótesis de errores:** se basan en la experiencia de dónde aparecen los errores más habituales en las fases de análisis y desarrollo. Los errores se buscan dónde se prevé que suelen presentarse. Es un método fuertemente dependiente de la experiencia.

El rango de posibles casos de prueba es tan grande que es casi imposible que puedas probarlos todos, por lo que hay que seleccionar un número de casos de prueba limitado. ¿Cuáles y cuántos?

Con la técnica de las particiones de equivalencia se consigue dividir todas las entradas en clases de equivalencia y se escogen diferentes casos de prueba de acuerdo a cada clase.

Dos casos de prueba son equivalentes si se espera que el programa los procese de la misma forma.

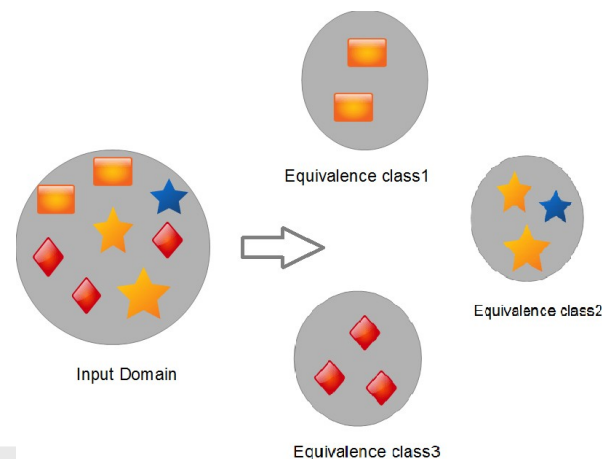
Una clase de equivalencia se obtiene a partir de las condiciones de entrada:

- Un valor específico
- Un rango de valores
- Un conjunto de valores relacionados
- Alguna condición lógica

Cada condición de entrada generará una o varias clases válidas y una o varias inválidas.

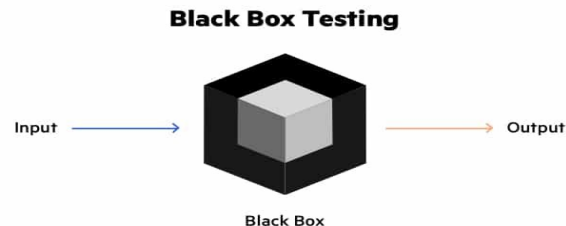
Cuando estemos haciendo casos de prueba, tendremos que tener en cuenta no sólo la

Entrada, sino también los distintos valores que puede presentar la salida!



Cosas a tener en cuenta:

- La creación de particiones de equivalencia no siempre define el número de casos de prueba.
- Si estamos probando un lenguaje fuertemente tipado (es decir, cuando asignamos un tipo a una variable no podemos asignar valores que no sean de ese tipo) anula las clases de equivalencia que comprueban el tipo de dato de la entrada... lo que es una pena que JavaScript sea un lenguaje débilmente tipado y que no de muchos errores al trabajar con diferentes tipos de datos
- El uso de fuerza bruta nos permite cubrir muchos casos de prueba y asegurarnos de que nuestro sistema funcione bien, pero es poco práctico cuando nuestro sistema tiene una gran cantidad de datos de entrada, ya que el número de casos de prueba aumenta de manera exponencial.
- Por mucho que no nos guste, a veces estaremos atascados en una situación que requiera el uso del testeo de caja negra:
- No hay acceso al código (o incluso si lo hubiese, no eres capaz de entenderlo)
- No tienes acceso a una especificación que explique el diseño del software
- No conoces al programador que lo ha hecho.



Usado junto a la técnica anterior, consiste en crear casos de prueba usando los valores límites que hay entre los rangos de los valores de entrada.

- Los límites suelen ser carne de error (usar $<$ en vez de $<=$...)
- A veces los requisitos no dejan claro cuál es el límite
- A veces representa límites en el código -> ¿Mi array es estático, dinámico, su tamaño puede cambiar...?

Por eso cuando se escogen valores de una clase de equivalencia se usan los valores que es más probable que causen un error, es decir, aquellos que se encuentren en los extremos en vez de en la parte central de un rango de valores.

Por ejemplo, para $N \rightarrow 18 < N < 30$.

¿Cuáles serán los valores límite?



Invalid test cases	Valid test cases	Invalid test cases
11, 13, 14, 15, 16, 17	18, 19, 24, 27, 28, 30	31, 32, 36, 37, 38, 39



Esta técnica funciona bien cuando el programa a tratar trabaja con variables independientes y valores físicos.

Si, por ejemplo, tenemos que testear una aplicación que trabaje con fecha (MES AÑO DIA) con el análisis de valores límite nos podríamos dejar casos tales como:

- Los días de febrero
- Los años bisiestos

Los casos de prueba obtenidos pueden ser, de cierto modo, rudimentarios ya que se crean con poca imaginación.

Este tipo de testing tienen mucha utilidad cuando trabajamos con cantidades físicas:

- Velocidades
- Ángulos
- Temperaturas

Si por ejemplo trabajamos con códigos postales, ¿Qué diferencia hay entre usar 00000 y 99999?

4. Testing

Pruebas de caja negra usando clases de equivalencia y valores límite

Veamos un ejemplo:

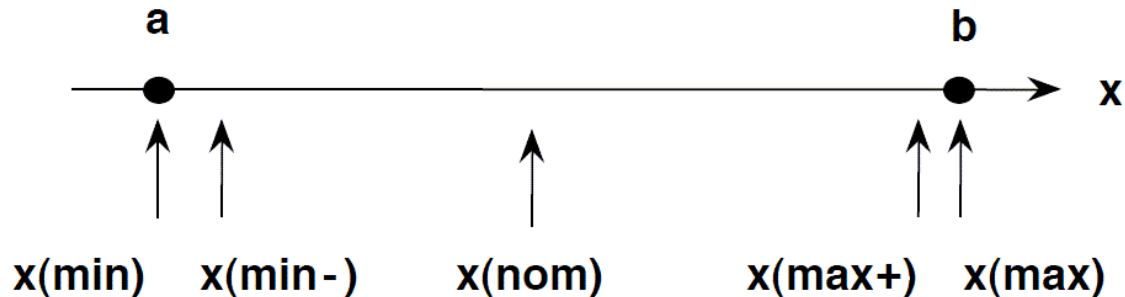
Un programa recibe como entrada un número entero y positivo de mínimo 2 cifras y de máximo 9 cifras y devuelve el número resultante de invertir sus cifras.

Si no se introduce un valor acorde a lo descrito (por ejemplo: floats y/o chars, valores fuera de rango, etc.), el programa devolverá el valor “error”. Generar la tabla de clases de equivalencia y los casos de prueba (así como el análisis de valores límite).

Ejemplos:

12345 -> 54321

10 -> 01



Se os ha encomendado la función ***int encontrarValorArray(int[] arrayBuscar, int valor)***, que devolverá la posición del elemento **valor** en **arrayBuscar**, si existe, y -1 en caso contrario.

Si la longitud del array es de más de 5 elementos, entonces el programa devolverá un error (**SORRY ARRAY IS TOO LONG**).

Si la longitud es menor de 2 elementos, devolverá un error (**SORRY MAN THIS IS TOO SMALL**).

Usando la técnica de las clases de equivalencia y de los valores límite, generar los casos de pruebas asociado para asegurarnos de su correcto funcionamiento.

Por ejemplo:

`encontrarValorArray([1,3,4,5],2) -> -1`

`encontrarValorArray([1,5],5) -> 1`

En resumen:

- Son técnicas antiguas (década de los 60) pero siguen teniendo relevancia hoy en día
- Nos ayudan a obtener una capacidad de abstracción lógica mejor.
- Son herramientas muy útiles, pero hay que usarlas con cabeza, estudiando los resultados y recalibrando los casos de prueba obtenidos
- No hay que enfocarse sólo en las entradas y en las salidas, sino también en cualquier tipo de condición que podamos observar en nuestro software.
- Es compatible con una metodología ágil!

Pruebas de caja blanca

En las pruebas de caja blanca se conoce o se tiene en cuenta el código que quiere probarse. También se les denomina *clear box testing* porque la persona que las realiza está en contacto con el código fuente. **Su objetivo es probar el código, cada uno de sus elementos.** Algunos tipos son:

- ✓ **Pruebas de cubrimiento:** es comprobar que todas las funciones, sentencias, decisiones, y condiciones, se van a ejecutar.
- ✓ **Pruebas de condiciones:** son necesarios varios casos de prueba ya que será necesario probar cada una de las posibles combinaciones de nuestros condicionales.
- ✓ **Pruebas de bucles:** los bucles son estructuras repetitivas, así por lo cual las pruebas serán repetir un número especial de veces dicho bucle
 - ✓ Repetir el máximo, máximo -1 y máximo +1 veces el bucle para ver si el resultado del código es el esperado
 - ✓ Repetir el bucle cero y una vez
 - ✓ Repetir el bucle un número determinado de veces
- **Pruebas de caminos:** Establece que se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final. La ejecución de este conjunto de sentencias, se conoce como camino.

Pruebas de caja blanca: Prueba del camino básico

La finalidad de esta técnica es poder diseñar casos de prueba que aseguren la ejecución de todos los posibles **caminos** de ejecución por lo menos una vez.

Un camino (o **path**) es una ejecución de una secuencia de instrucciones desde el punto de inicio hasta el punto final. Un programa informático recibe unas entradas y genera una salidas, por lo que es natural tener que ejecutar tantos caminos como sea posible para asegurar el correcto funcionamiento.

Para visualizar los diferentes caminos, **se traduce el diseño secuencial a un grafo de flujo**, el cual está formado por nodos y aristas:

- Los **nodods** representan uno o más instrucciones secuenciales.
- Las **aristas** representan el flujo de control, es decir, el siguiente nodo que puede ser ejecutado. Si un nodo tiene más de un arista, la elección de una u otra dependerá de diferentes condiciones.
- Las áreas delimitadas por aristas y nodos las llamamos **regiones**.

4. Testing

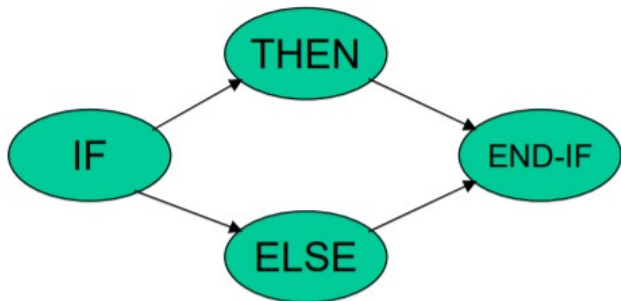
Pruebas de caja blanca: Prueba del camino básico



SECUENCIA



```
Instrucción 1  
Instrucción 2  
.....  
Instrucción n
```

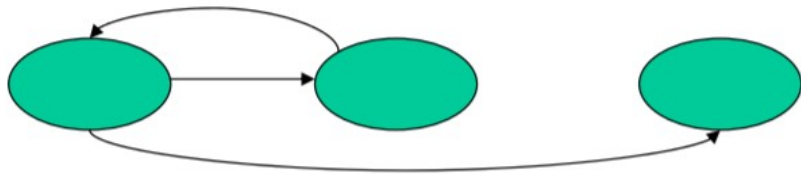


SI,..SINO..



CONDICIONAL

```
Si <condición> Entonces  
    <Instrucciones>  
Si no  
    <Instrucciones>  
Fin si
```

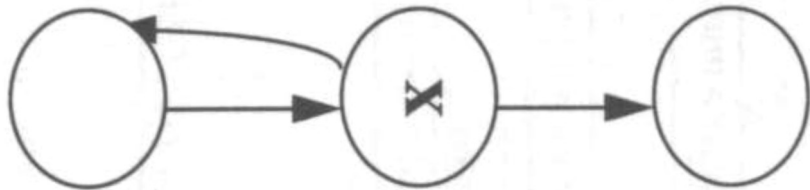


WHILE



HACER MIENTRAS

```
Mientras <condición> Hacer  
    <Instrucciones>  
Fin mientras
```

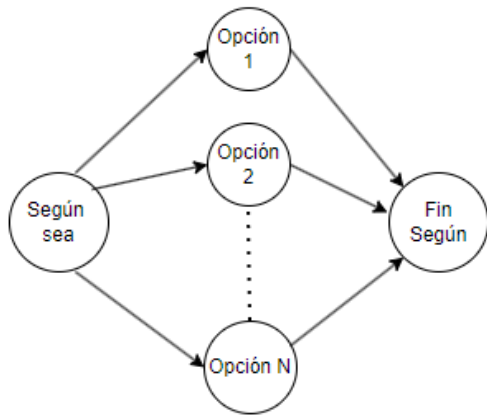


REPETIR HASTA

Repetir

<Instrucciones>

Hasta que <condición>



CONDICIONAL MÚLTIPLE

Según sea <variable> Hacer

Caso opción 1:

<Instrucciones>

Caso opción 2:

<Instrucciones>

Caso opción 3:

<Instrucciones>

Otro Caso:

<Instrucciones>

Fin según

Pruebas de caja blanca: Prueba del camino básico

Esta técnica nos permite obtener una medida de la **complejidad lógica**, que determina la cota superior del número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez.

De eso se infiere que **cuanto menor es la complejidad, menor es la cantidad de pruebas necesarias para el método en cuestión.**

Una vez calculada la complejidad ciclomática de un fragmento de código, se puede determinar el riesgo que supone utilizando los rangos definidos en la siguiente tabla:

Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, riesgo muy alto

La complejidad ciclomática es una medida del software que aporta una valoración cuantitativa de la complejidad lógica de un programa.

Dentro del contexto del método de pruebas del camino básico **define el número de caminos independientes de un programa.**

Por camino independiente se entiende aquel que introduce un nuevo conjunto de sentencias o una nueva condición.

En términos del grafo, por una arista que no haya sido recorrida antes. Nos da una cota o límite superior para el número de casos de prueba.

Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición, respecto a los caminos existentes.

En términos del diagrama de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino.

No se considera un camino independiente si es una combinación de caminos ya especificados y no recorre ninguna arista nueva.

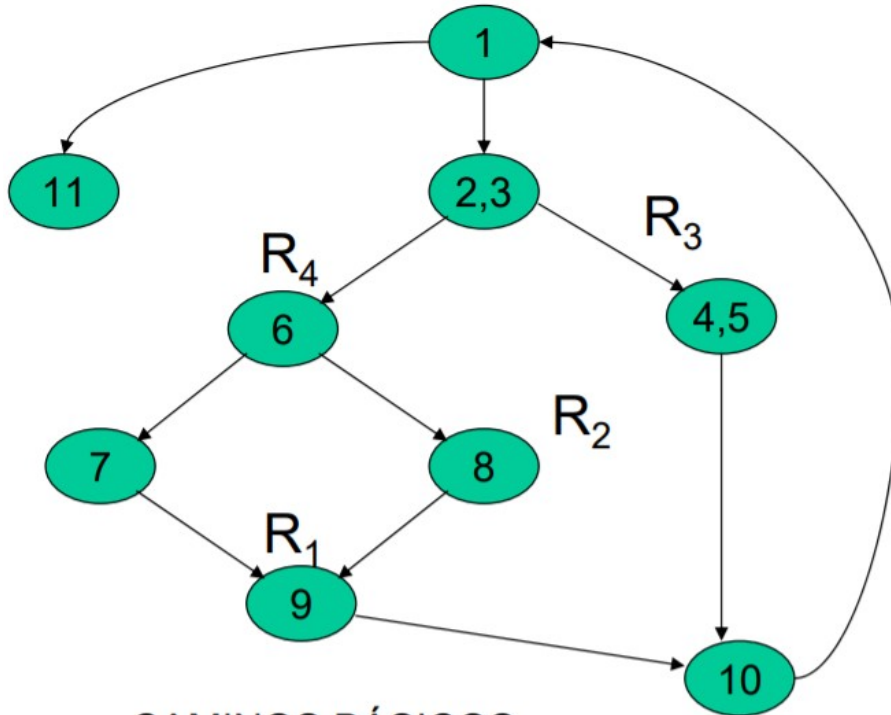
En la identificación de los distintos caminos de un programa para probar, **se debe tener en cuenta que cada nuevo camino debe tener el mínimo número de sentencias nuevas o condiciones nuevas respecto a los que ya existen.**

Cada camino no puede recorrer la misma arista dos veces (en el caso de los bucles)

Existen varias formas de calcular la complejidad ciclomática de un programa a partir de un grafo de flujo:

- **Según el número de regiones del grafo:** El número de regiones del grafo coincide con la complejidad ciclomática, $V(G)$.
- $V(G) = A - N + 2$
 - Donde **A** es el número de aristas y **N** es el número de nodos contenidos en el grafo.
- $V(G) = nps + 1$
 - Donde **nps** es el número de nodos con predicado simple (aquel nodo de los que parten dos caminos, es decir, que representa alguna estructura condicional) contenidos en el grafo.

¿Cuáles son los caminos básicos de este grafo? ¿Y su complejidad ciclomática?

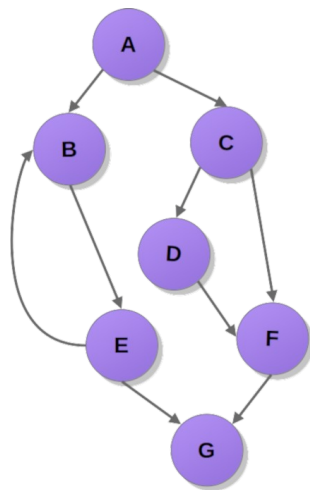


$$V(g) = A - N + 2 = 11 - 9 + 2 = 4$$

$$V(g) = nps + 1 = 3 + 1 = 4$$

Ejemplo. Dado el siguiente código en Python obtenemos su grafo de flujo.

```
1. import random
2. num = random.randint(1, 10)
3. print("Inicia el programa")
4. if num > 3:
5.     for i in range(num):
6.         print("Hola")
7. else:
8.     if num == 2:
9.         print("El número es 2")
10.    print("El número es menor a 3")
11. print("Fin del programa")
```



En este ejemplo el nodo **A** representa las líneas 1-4 del código, que incluyen hasta la condición `if`. El nodo **B** representa la línea 5, donde se realiza la iteración `for`. El nodo **E** es la línea 6, que es la que se repite dentro del bucle `for`. El nodo **C** incluye las líneas 7 y 8, donde se realiza otro `if`.

Por otro lado, el nodo **D** representa la línea 9 y el nodo **F** la 10. **G** es el único final del sistema que representa la línea 11 ejecutando la última línea de código, por la que se pasa independientemente del camino elegido.

`range(num)`: crea una serie que empieza en 0 y termina en `num`
`Random.randint(a,b)`: retorna un aleatorio `a <= num <= b`

¿Cuál sería la complejidad ciclomática de este grafo?



Diseñar el conjunto de casos de prueba mediante el método de la complejidad ciclomática para el siguiente código:

```
int contar_letras(char cadena[10], char letra)
{
    int contador=0, n=0, lon;
    lon = strlen(cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra) n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```

Diseñar el conjunto de casos de prueba mediante el método de la complejidad ciclomática para el siguiente código:

```
if (a == 50)
{
    if(b == c)
    {
        a = c;
    }
    else
    {
        b = c;
    }
}
System.out.println("El valor de a es" + a)
```

Diseñar el conjunto de casos de prueba mediante el método de la complejidad ciclomática para el siguiente código:

```
read (x,y);
for (i = 1; i <= 2; i++){
    print ("Pau Gasol");
}
print ("Marc Gasol");
if (y < 0) {
    print ("Adrià Gasol");
}
else{
    print (x);
}
```

Diseñar el conjunto de casos de prueba mediante el método de la complejidad ciclomática para el siguiente código:

```
public int aMethod(  
    boolean a, boolean b,  
    boolean c) {  
    int ret = 0;  
  
    if (a && b) {  
        ret = 1;  
    } else if (c) {  
        ret = 2;  
    }  
    return ret;  
}
```

TABLE I. COMPARISON BETWEEN THREE FORMS OF TESTING TECHNIQUES [6] [7]

S. No.	Black Box Testing	Grey Box Testing	White Box Testing
1.	Analyses fundamental aspects only i.e. no proved edge of internal working	Partial knowledge of internal working	Full knowledge of internal working
2.	Granularity is low	Granularity is medium	Granularity is high
3.	Performed by end users and also by tester and developers (user acceptance testing)	Performed by end users and also by tester and developers (user acceptance testing)	It is performed by developers and testers
4.	Testing is based on external exceptions – internal behaviour of the program is ignored	Test design is based on high level database diagrams, data flow diagrams, internal states, knowledge of algorithm and architecture	Internal are fully known
5.	It is least exhaustive and time consuming	It is somewhere in between	Potentially most exhaustive and time consuming
6.	It can test only by trial and error method	Data domains and internal boundaries can be tested and over flow, if known	Test better: data domains and internal boundaries
7.	Not suited for algorithm testing	Not suited for algorithm testing	It is suited for algorithm testing (suited for all)

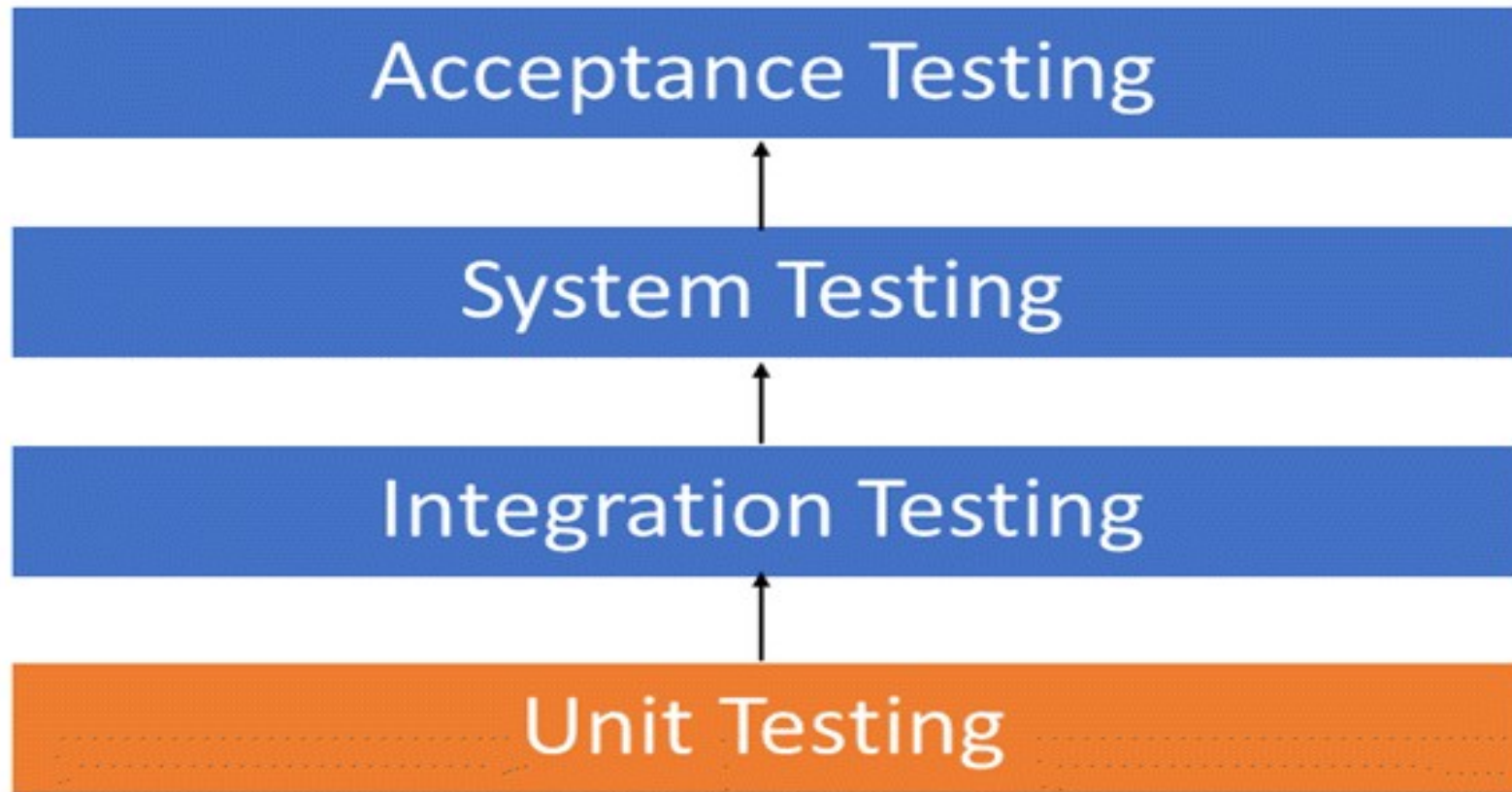
Ehmer, Mohd & Khan, Farmeena. (2012). A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. International Journal of Advanced Computer Science and Applications. 3. 10.14569/IJACSA.2012.030603.



Planificación de pruebas

La planificación de las pruebas es un punto importante en la toma de decisiones de un proyecto. Qué tipo de prueba y cuando van a realizarse son preguntas que hay que tener en cuenta desde el principio.

- **Pruebas unitarias:** se comienza por las pruebas por separado de cada módulo del software.
- **Pruebas de integración:** a partir del esquema de diseño, los módulos probados se vuelven a probar combinados para probar sus interfaces.
- **Prueba de sistema:** se procede a realizar la prueba de un sistema integrado de hardware y software para comprobar si cumple los requisitos funcionales.
- **Prueba de aceptación:** es la prueba para determinar si se cumplen los requisitos de aceptación marcados por el cliente.





Las pruebas unitarias, **tienen por objetivo probar el correcto funcionamiento de un módulo de código**. El fin que se persigue, **es que cada módulo funciona correctamente por separado**.

Posteriormente, con la prueba de integración, se podrá asegurar el correcto funcionamiento del sistema.

Una unidad es la parte de la aplicación más pequeña que se puede probar. En programación orientada a objetos, una unidad es normalmente un método.

Con las pruebas unitarias se **debe probar todas las funciones o métodos no triviales de forma que cada caso de prueba sea independiente del resto**.

En el diseño de los casos de pruebas unitarias, habrá que tener en cuenta los siguientes requisitos:

- **Automatizable:** no debería requerirse una intervención manual.
- **Completas:** deben cubrir la mayor cantidad de código.
- **Repetibles** o Reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- **Independientes:** la ejecución de una prueba no debe afectar a la ejecución de otra.
- **Profesionales:** las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Las pruebas individuales nos proporcionan cinco ventajas básicas:

- 1) **Fomentan el cambio:** Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
- 2) **Simplifica la integración:** Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
- 3) **Documenta el código:** Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
- 4) **Separación de la interfaz y la implementación:** Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
- 5) **Los errores están más acotados** y son más fáciles de localizar: dado que tenemos pruebas unitarias que pueden desenmascararlos



Pruebas de integración

Las pruebas de integración están totalmente ligadas a la forma prevista de integrar los distintos componentes del software hasta contar con el producto global que debe entregarse. Así, las pruebas de integración implican una progresión ordenada de pruebas que parten desde los componentes (módulos) y que culmina en el sistema completo. **Su objetivo fundamental es la prueba de las interfaces (flujo de datos) entre componentes o módulos.**

Las pruebas de integración podemos clasificarlas como:

- **Integración incremental.** Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados. Puede ser ascendente (se comienza por los módulos hoja) o descendente (comienza por el módulo raíz)
- **Integración no incremental.** Se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo

Habitualmente las pruebas de unidad y de integración se solapan y mezclan en el tiempo



La prueba de sistema es el proceso de prueba de un sistema integrado de hardware y software para comprobar si cumple los requisitos especificados, es decir:

- Se cumplen todos los requisitos funcionales establecidos.
- El funcionamiento y rendimiento de las interfaces hardware, software y de usuario
- La adecuación de la documentación de usuario
- El rendimiento y respuesta en condiciones límite y de sobrecarga.

Para la generación de casos de prueba de sistema se utilizan técnicas de caja negra.



Pruebas de aceptación

El objetivo que se pretende conseguir con estas pruebas es comprobar si el producto está listo para ser implantado para el uso operativo en el entorno del usuario.

Las características principales de esta prueba son:

- **Participación del usuario:** El usuario debe ser el que realice las pruebas, ayudado por personas del equipo de pruebas, siendo deseable, que sea él mismo quien aporte los casos de prueba.
- Está enfocada a **probar los requisitos del usuario especificados**, adoptados en forma de criterios de aceptación, que deben ser preferentemente objetivos y medibles. Estos criterios se fijan simultáneamente a los requisitos del sistema, al inicio del ciclo de desarrollo.

Pruebas unitarias en C#

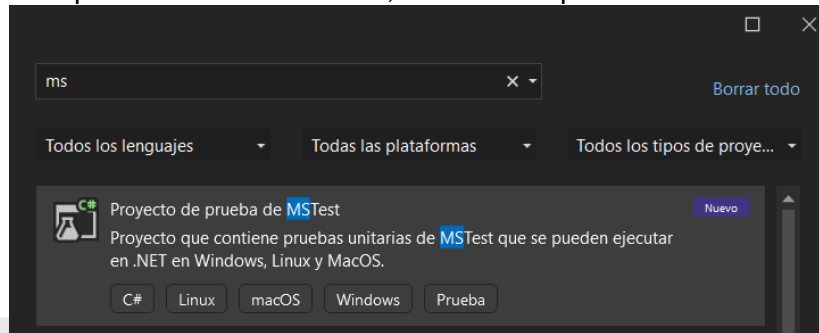
Como ya hemos visto, y de forma resumida, hacer test unitarios consiste en probar partes individuales de un software y comprobar su funcionamiento.

De forma general, ***estos casos de prueba serán lanzados como funciones que evaluarán la ejecución de la unidad a testear y determinarán si la salida del mismo es la esperada o no.***

En los diversos lenguajes hay herramientas que permiten la automatización de los test unitarios, en C# hay varios: Nunit, Xunit, MsTest... pero todos comparten una sintaxis similar, basada en el uso de ***etiquetas*** para identificar los diferentes elementos de nuestro test.

El más básico, es la etiqueta ***[TestClass]***, la cual indica una clase que se encargará de lanzar tests, especificados como ***[TestMethod]***

Para poder lanzar estos tests, tendremos que añadir un nuevo proyecto a nuestra solución. Buscad ***MSTest***.



4. Testing

Pruebas unitarias en C#

Vamos a ver un ejemplo de test
con esta clase sencilla

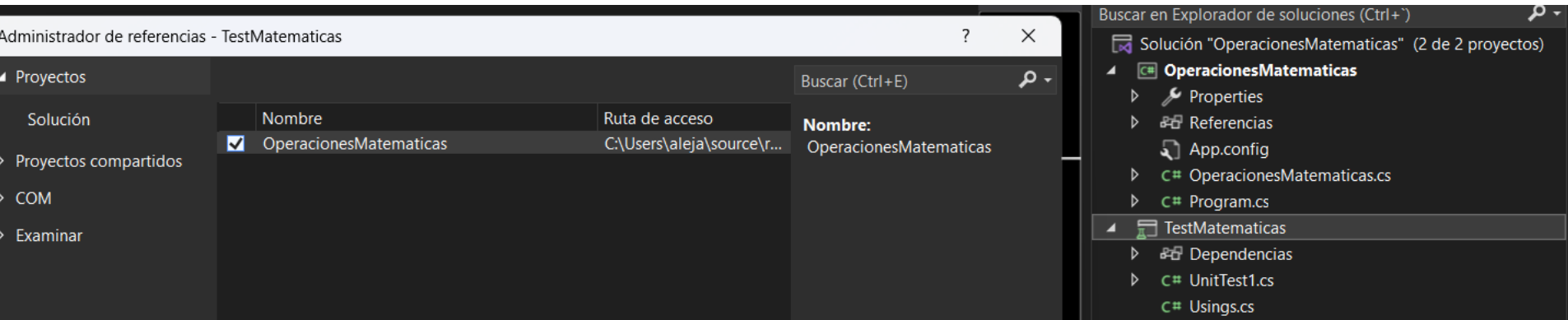
```
OperacionesMatematicas
{
1  namespace Clases;
   3 referencias
2  public class OperacionesMatematicasClass
3  {
   2 referencias | 0/1 pasando
4      public static double sumar(double num1, double num2)
5      {
6          return num1 + num2;
7      }
8
   0 referencias
9      public double restar(double num1, double num2)
10     {
11         return num1 - num2;
12     }
13
   0 referencias
14     public double dividir(double num1, double num2)
15     {
16         return num1 / num2;
17     }
18     public double multiplicar(double num1, double num2)
19     {
   1 referencia | 0/1 pasando
20         return num1 + num2; // debería multiplicar
21     }
22 }
```

4. Testing

Pruebas unitarias en C#

Una vez creamos el proyecto de test, tendremos dos proyectos en nuestra solución, por lo que tendremos que añadir referencias de un proyecto a otro para que puedan acceder a las clases.

Para ello, seleccionad vuestro proyecto de test, y dadle **click derecho > add > Project reference**



Vamos a ver un ejemplo de test

```
namespace TestMatematicas
{
    [TestClass]
    0 referencias
    public class UnitTest1
    {
        [TestMethod]
        0 referencias
        public void probarSuma()
        {
            int a, b;
            a = b = 10;
            double resultado = OperacionesMatematicasClass.sumar(a, b);
            Assert.AreEqual(20, resultado);
        }
    }
}
```


Los test que corramos podrán ser válidos o inválidos, en base a una serie de condiciones que estableceremos. Estas condiciones representan que la salida del test coincida con lo esperado.

Para ello, podemos usar la [clase Assert](#), la cual contiene una gran cantidad de funciones que nos devolverán **verdadero** o **falso** dependiendo de si los valores pasados como parámetros son iguales o no.

La clase Assert permite comparar diferentes tipos de datos, saber si un elemento existe en una lista, si dos listas son iguales, si el método ha lanzado una excepción...

```
Assert.AreEqual(28, _actualFuel); // Tests whether the specified values are equal.
Assert.AreNotEqual(28, _actualFuel); // Tests whether the specified values are unequal. Same as AreEqual for numeric values.
Assert.AreSame(_expectedRocket, _actualRocket); // Tests whether the specified objects both refer to the same object
Assert.AreNotSame(_expectedRocket, _actualRocket); // Tests whether the specified objects refer to different objects
Assert.IsTrue(_isThereEnoughFuel); // Tests whether the specified condition is true
Assert.IsFalse(_isThereEnoughFuel); // Tests whether the specified condition is false
Assert.IsNull(_actualRocket); // Tests whether the specified object is null
Assert.IsNotNull(_actualRocket); // Tests whether the specified object is non-null
Assert.IsInstanceOfType(_actualRocket, typeof(Falcon9Rocket)); // Tests whether the specified object is an instance of the expected type
Assert.IsNotInstanceOfType(_actualRocket, typeof(Falcon9Rocket)); // Tests whether the specified object is not an instance of type

StringAssert.Contains(_expectedBellatrixTitle, "Bellatrix"); // Tests whether the specified string contains the specified substring
StringAssert.StartsWith(_expectedBellatrixTitle, "Bellatrix"); // Tests whether the specified string begins with the specified substring
StringAssert.Matches("(281)388-0388", @"(?:\d{3})\d{3}-\d{4}"); // Tests whether the specified string matches a regular expression
StringAssert.DoesNotMatch("(281)388-0388", @"(?:\d{3})\d{3}-\d{3}-\d{4}"); // Tests whether the specified string does not match a regular expression

CollectionAssert.AreEqual(_expectedRockets, _actualRockets); // Tests whether the specified collections have the same elements in the same order and quantity
CollectionAssert.AreNotEqual(_expectedRockets, _actualRockets); // Tests whether the specified collections does not have the same elements or the elements are in a different order
CollectionAssert.AreEqual(_expectedRockets, _actualRockets); // Tests whether two collections contain the same elements.
CollectionAssert.AreNotEqual(_expectedRockets, _actualRockets); // Tests whether two collections contain different elements.
CollectionAssert.AllItemsAreInstancesOfType(_expectedRockets, _actualRockets); // Tests whether all elements in the specified collection are instances of the specified type
CollectionAssert.AllItemsAreNotNull(_expectedRockets); // Tests whether all items in the specified collection are non-null
CollectionAssert.AllItemsAreUnique(_expectedRockets); // Tests whether all items in the specified collection are unique
CollectionAssert.Contains(_actualRockets, falcon9); // Tests whether the specified collection contains the specified element
CollectionAssert.DoesNotContain(_actualRockets, falcon9); // Tests whether the specified collection does not contain the specified element
CollectionAssert.IsSubsetOf(_expectedRockets, _actualRockets); // Tests whether one collection is a subset of another collection
CollectionAssert.IsNotSubsetOf(_expectedRockets, _actualRockets); // Tests whether one collection is not a subset of another collection

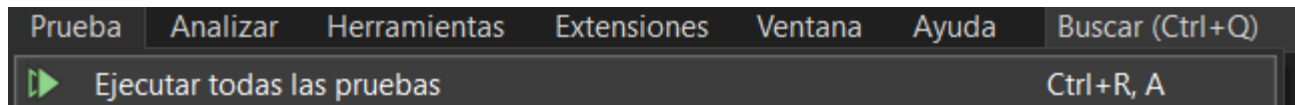
Assert.ThrowsException<ArgumentNullException>(() => new Regex(null)); // Tests whether the code specified by delegate throws exact given exception of
```

4. Testing

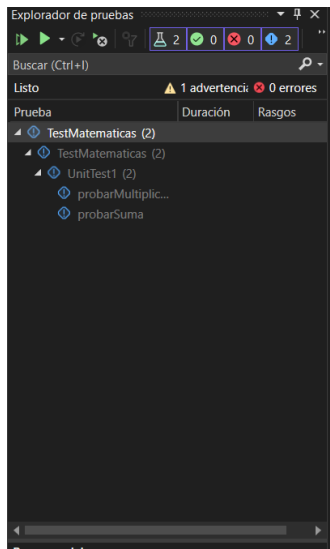
Pruebas unitarias en C#

Vamos a afinar un poco más...

Podréis lanzar las pruebas ya sea desde **Prueba > Run All Test...**



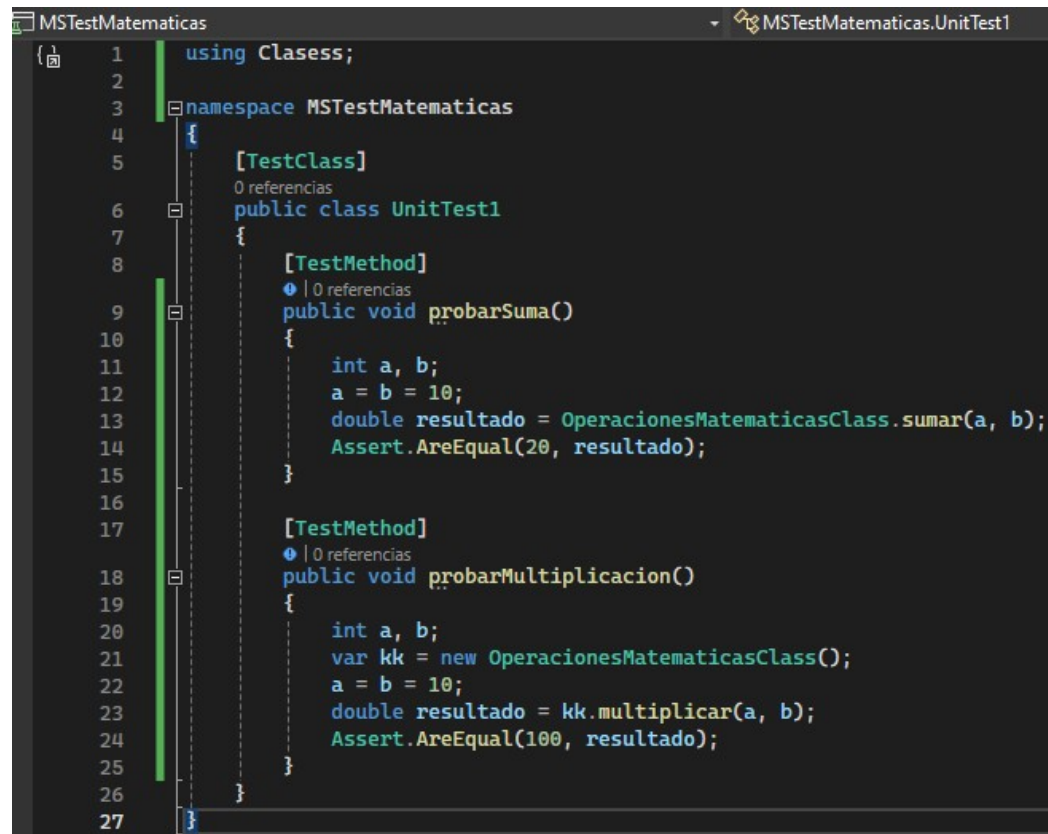
... desde la ventana de pruebas (**Ver > Explorador de pruebas**)



4. Testing

Pruebas unitarias en C#

Vamos a lanzar
estos test...



The screenshot shows a Visual Studio code editor with a C# file named `MSTestMatematicas.UnitTest1`. The code defines a namespace `MSTestMatematicas` containing a `TestClass` with two test methods: `probarSuma()` and `probarMultiplicacion()`. Both methods use `Assert.AreEqual` to verify the results of operations performed by `OperacionesMatematicasClass`.

```
1 using Classes;
2
3 namespace MSTestMatematicas
4 {
5     [TestClass]
6     public class UnitTest1
7     {
8         [TestMethod]
9         public void probarSuma()
10        {
11            int a, b;
12            a = b = 10;
13            double resultado = OperacionesMatematicasClass.sumar(a, b);
14            Assert.AreEqual(20, resultado);
15        }
16
17        [TestMethod]
18        public void probarMultiplicacion()
19        {
20            int a, b;
21            var kk = new OperacionesMatematicasClass();
22            a = b = 10;
23            double resultado = kk.multiplicar(a, b);
24            Assert.AreEqual(100, resultado);
25        }
26    }
27 }
```

Pruebas unitarias en C#

Vamos a lanzar estos test...

Prueba	Duración	Rasgos	Mensaje de error
TestMatematicas (2)	69 ms		
TestMatematicas (2)	69 ms		
UnitTest1 (2)	69 ms		
probarMultiplic...	69 ms		Error de Assert.AreEqual. Se...
probarSuma	< 1 ms		

Resumen de los detalles de la prueba
probarMultiplicacion
Origen: UnitTest1.cs línea 16
Duración: 69 ms
Mensaje:
Error de Assert.AreEqual. Se esperaba <100>, pero es <20>.
Seguimiento de la pila:
UnitTest1.probarMultiplicacion() línea 21

Los test que no se pasan devolverán un error informando de la salida esperada y la salida obtenida. Esta información nos indicará que hay algún error en nuestro código.

4. Testing

Pruebas unitarias en C#

Además de las etiquetas básicas, también tenemos otras etiquetas, como la etiqueta **`[Ignore]`**, la cual nos permitirá ignorar un test

The screenshot displays the Visual Studio interface during a unit test run. On the left, the 'Test Results' window shows a list of tests: 'TestMatematicas (3)' (41 ms), 'TestMatematicas (3)' (41 ms), 'UnitTest1 (3)' (41 ms), 'noMeHacenCaso' (41 ms), 'probarMultiplic...' (41 ms), and 'probarSuma' (< 1 ms). The 'noMeHacenCaso' test is highlighted with a yellow warning icon. Below this, the 'Resumen de los detalles de la prueba' section shows the origin: 'Origen: UnitTest1.cs línea 26'.

The main editor shows the source code of 'UnitTest1.cs'. The code includes three test methods:

```
public class UnitTest1
{
    [TestMethod]
    public void probarSuma()
    {
        a = b = 10;
        double resultado = OperacionesMatematicasClass.sumar(a, b);
        Assert.AreEqual(20, resultado);
    }

    [TestMethod]
    public void probarMultiplicacion()
    {
        int a, b;
        a = b = 10;
        double resultado = OperacionesMatematicasClass.multiplicar(a, b);
        Assert.AreEqual(100, resultado);
    }

    [TestMethod]
    [Ignore]
    public void noMeHacenCaso()
    {
        throw new IndexOutOfRangeException();
    }
}
```

The `[Ignore]` attribute on the `noMeHacenCaso` method is highlighted with a red box.

¿Por qué voy a querer ignorar un test? **`[Ignore]`**

[Lectura recomendada](#)

Pruebas unitarias en C#

Cuando sepamos que cierto caso de prueba debe devolver una excepción (dividir por cero, intentar pasar de tipos, pasarnos de longitud de un array...) podemos usar la etiqueta `[ExpectedException(typeof(ExcepcionACapturar))]`, donde **ExcepcionACapturar** es el tipo de excepción que esperamos.



The screenshot displays the Visual Studio IDE. On the left, the 'Test Explorer' pane shows a test named 'TestQueEsperaUnaExcepcion' with a green checkmark, indicating it passed. Below the test name, it shows the origin as 'UnitTest1.cs línea 33' and the duration as '5 ms'. The main editor area shows the source code for the test method. The method is decorated with the attribute `[ExpectedException(typeof(IndexOutOfRangeException))]`, which is highlighted with a red rectangle. The method signature is `public void TestQueEsperaUnaExcepcion()`, also highlighted with a green dashed box. The method body contains two `throw new IndexOutOfRangeException();` statements, with the second one highlighted with a red underline. A vertical timeline on the left of the code editor shows the execution flow of the test.

```
[TestMethod]
[ExpectedException(typeof(IndexOutOfRangeException))]
public void TestQueEsperaUnaExcepcion()
{
    throw new IndexOutOfRangeException();

    throw new IndexOutOfRangeException();
}
```



Ejercicios

1. Crea una función ***public static int numMenosMedia(int[] array)***, que devuelva el número de elementos que tenga ***array*** con valores inferiores a la media de los valores del array.
2. Crea una función ***public static bool estaOrdenada(int[] array)*** que devolverá true, si el array está ordenado de forma descendente.
3. Crea una función ***public static bool comprobarPassword(String pass)*** que devolverá true si la contraseña pasada como parámetro cumple las siguientes condiciones:
 - Tiene entre 4 y 6 caracteres.
 - Tiene una minúscula y una mayúscula
 - Contiene un dígito
 - No contiene la palabra ***Hitler***.

Crea estos 3 ejercicios en una misma solución, y diseña casos de prueba ya sea usando técnicas de caja blanca o caja negra. Implementa dichos casos de prueba usando MSTest (**crea un proyecto de test para cada función**)



Bibliografía

- **Sommerville, I. (2011). *Software engineering (ed.)*. America: Pearson Education Inc.**
- **Myers, Glenford J., Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.**
- **Jorgensen, Paul C. *Software testing: a craftsman's approach*. CRC press, 2018. -> caja blanca**
- **Kaner, Cem, Sowmya Padmanabhan, and Douglas Hoffman. *The Domain Testing Workbook*. Context Driven Press, 2013. -> caja Negra**
- **Madhavi, Dr. "A White Box Testing Technique in Softwre Testing: Basis Path Testing." *Journal for Research* 1.04 (2016). -> caja blanca**