

Entornos de Desarrollo

05 Clean Code y TDD: Pruebas de Software

José Luis González Sánchez



Contenidos

¿Qué voy a aprender?



Contenidos

1. Introducción al test y pruebas de software. Estrategias
2. Herramientas de Depuración
3. Excepciones
4. Pruebas de Caja Blanca
5. Pruebas de Caja Negra
6. Principios del TDD
7. JUnit

Introducción al test y pruebas de software. Estrategias

Buscando el error en nuestro código



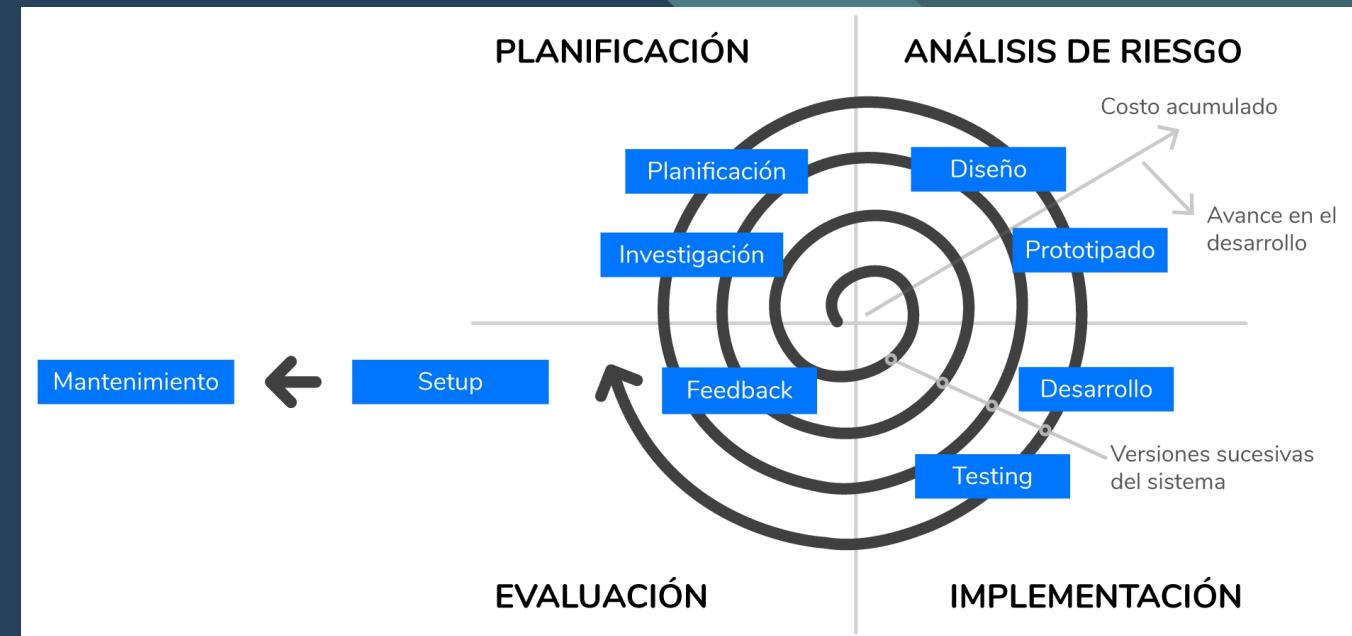
Introducción. Verificación y Validación

- Durante todo el proceso de desarrollo de software, desde la fase de diseño, en la implementación y una vez desarrollada la aplicación, es necesario realizar un **conjunto de pruebas**, que permitan verificar que el software que se está creando, es correcto y cumple con las especificaciones impuesta por el usuario.
- 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.
- Mediante la realización de pruebas se software, se van a realizar las tareas de **verificación y validación** del software.
- **La verificación es la comprobación que un sistema o parte de un sistema, cumple con las condiciones impuestas.** La verificación se comprueba si la aplicación se está construyendo correctamente.
- **La validación es el proceso de evaluación del sistema o de uno de sus componentes, para determinar si satisface los requisitos especificados.** Es decir, estamos haciendo aquello que nos han pedido que hagamos respecto a los requisitos del sistema.



Introducción. Verificación y Validación

- Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas.
- Siguiendo el Modelo en Espiral, las pruebas empezarían con la prueba de unidad, donde se analizaría el código implementado y seguiríamos en la prueba de integración, donde se prestan atención al diseño y la construcción de la arquitectura del software.
- El siguiente paso sería la prueba de validación, donde se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software.
- Finalmente se alcanza la prueba de sistema que verifica el funcionamiento total del software y otros elementos del sistema.





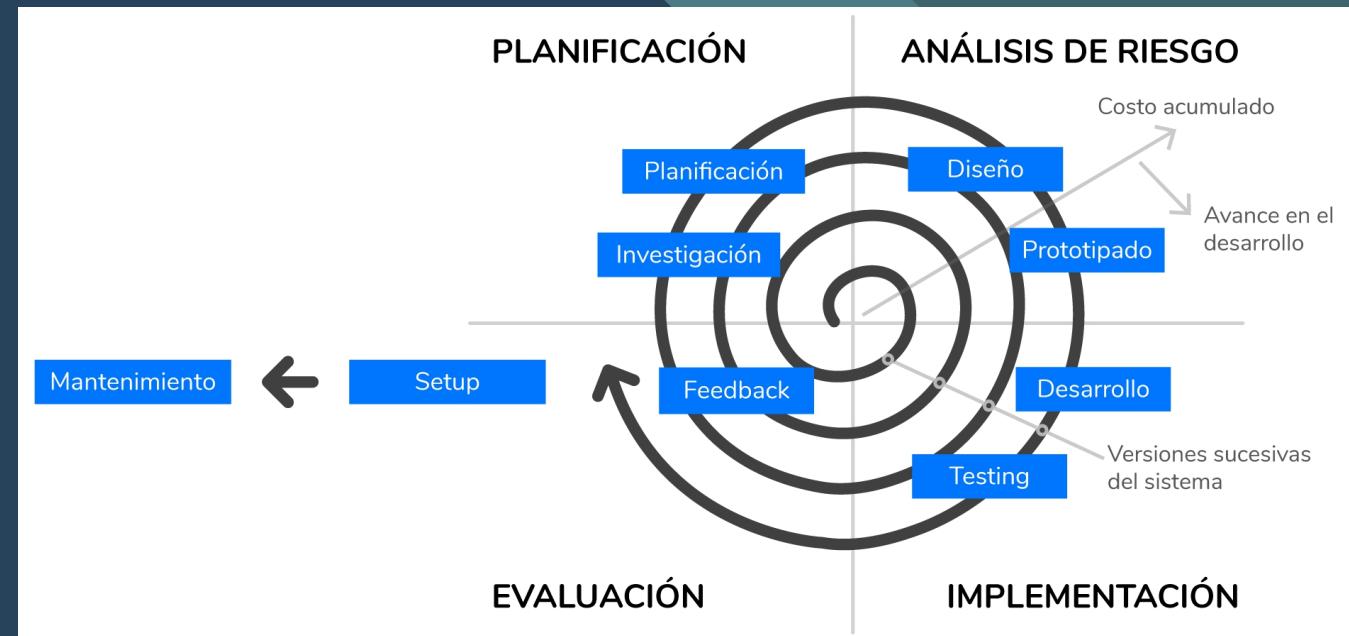
Introducción. Caso de Prueba

- En ingeniería del software, un caso de prueba (en inglés, test case) es un conjunto de condiciones o variables bajo las cuales se determinará si una aplicación, un sistema de software o una característica o comportamiento de estos resulta o no aceptable en base a los requisitos dados.
- Se debe realizar la prueba positiva de los requisitos y el otro debe realizar la prueba negativa.
- Si la aplicación es creada sin requisitos formales, entonces los casos de prueba se escriben basados en la operación normal de programas de una clase similar.
- Partimos siempre de una **entrada conocida y una salida esperada**, los cuales son formulados antes de que se ejecute la prueba. La entrada conocida debe probar una **pre condición** y la salida esperada debe probar una **postcondición**.
- Los casos de prueba escritos, incluyen una descripción de la funcionalidad que se probará, la cual es tomada ya sea de los requisitos, casos de uso o historias de usuario, y la preparación requerida para asegurarse de que la prueba pueda ser dirigida.
- Los casos de prueba escritos se recogen generalmente en una suite de pruebas.



Introducción. Estrategias

- Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas.
- Siguiendo el Modelo en Espiral, las pruebas empezarían con la prueba de unidad, donde se analizaría el código implementado y seguiríamos en la prueba de integración, donde se prestan atención al diseño y la construcción de la arquitectura del software.
- El siguiente paso sería la prueba de validación, donde se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software.
- Finalmente se alcanza la prueba de sistema que verifica el funcionamiento total del software y otros elementos del sistema.





Introducción. Pruebas de Unidad

- Se prueba cada unidad, módulo y métodos asociados con el I objetivo de eliminar errores en la interfaz y en la lógica interna. Se utilizan **técnicas de caja negra y caja blanca o herramientas de pruebas automáticas como JUnit, Jest, PHPUnit...**
- Probamos:
 - La interfaz del módulo, para asegurar que la información fluye adecuadamente
 - La estructuras de datos locales, para asegurar que mantienen su integridad durante la ejecución.
 - Las condiciones límite, para asegurar que funciona correctamente en los límites establecidos durante el proceso.
 - Todos los caminos independientes de las estructuras de control, con el fin de segurar que todas las sentencias se ejecutan al menos una vez.
 - Todos los caminos de manejo de errores.
- Ventajas:
 - Permiten saber si estamos revalorizando bien, integrando y nos ayudan a documentar.
 - Los errores están más acotados y son más fáciles de localizar.

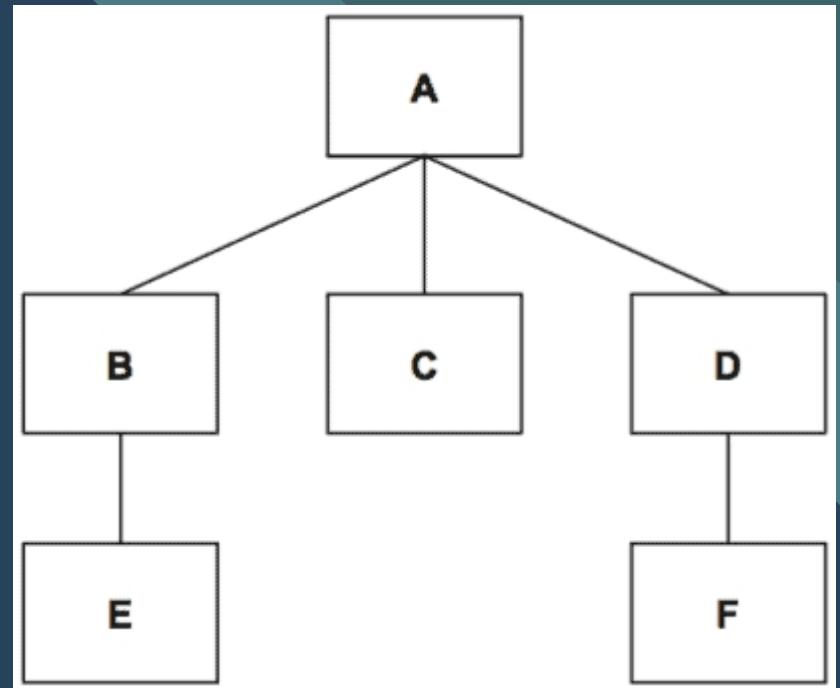


Introducción. Pruebas de Integración

- Probamos como interactúan los distintos módulos, es decir, no solo por separado, si no de forma conjunta.

Podemos hacerlo:

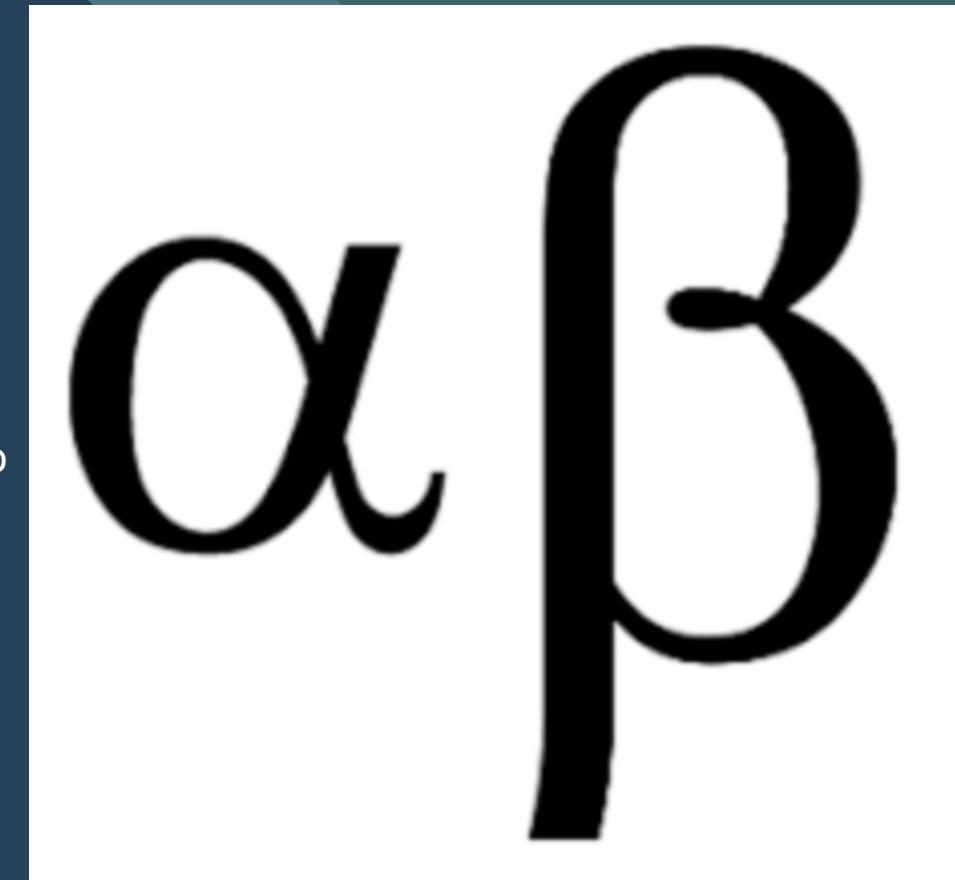
- No incremental o big bang: Se prueba cada módulo por separado y luego todos a la vez, lo cual no es muy recomendable porque aumenta la dificultad.
- Incremental. Se prueba segmento a segmento de manera Ascendente o Descendente. Ascendente probamos de los módulos de más bajo nivel a los de más alto nivel (hasta llegar al principal). Descendente, partimos del principal y nos movemos hacia los módulos más básicos o inferiores.





Introducción. Pruebas de Validación

- Consiste en probar el sistema de acuerdo con las expectativas del cliente
 - Pruebas Alfa: Se llevan a cabo por el cliente o usuario en el lugar de desarrollo bajo la observación del desarrollador que irá registrando errores y problemas de uso.
 - Pruebas Beta: se llevan a cabo por los usuarios finales del software en su lugar de trabajo. El desarrollador no está presente. El usuario o software registra todos los problemas en base a logs que son enviados al desarrollador para mejorar la versión final.





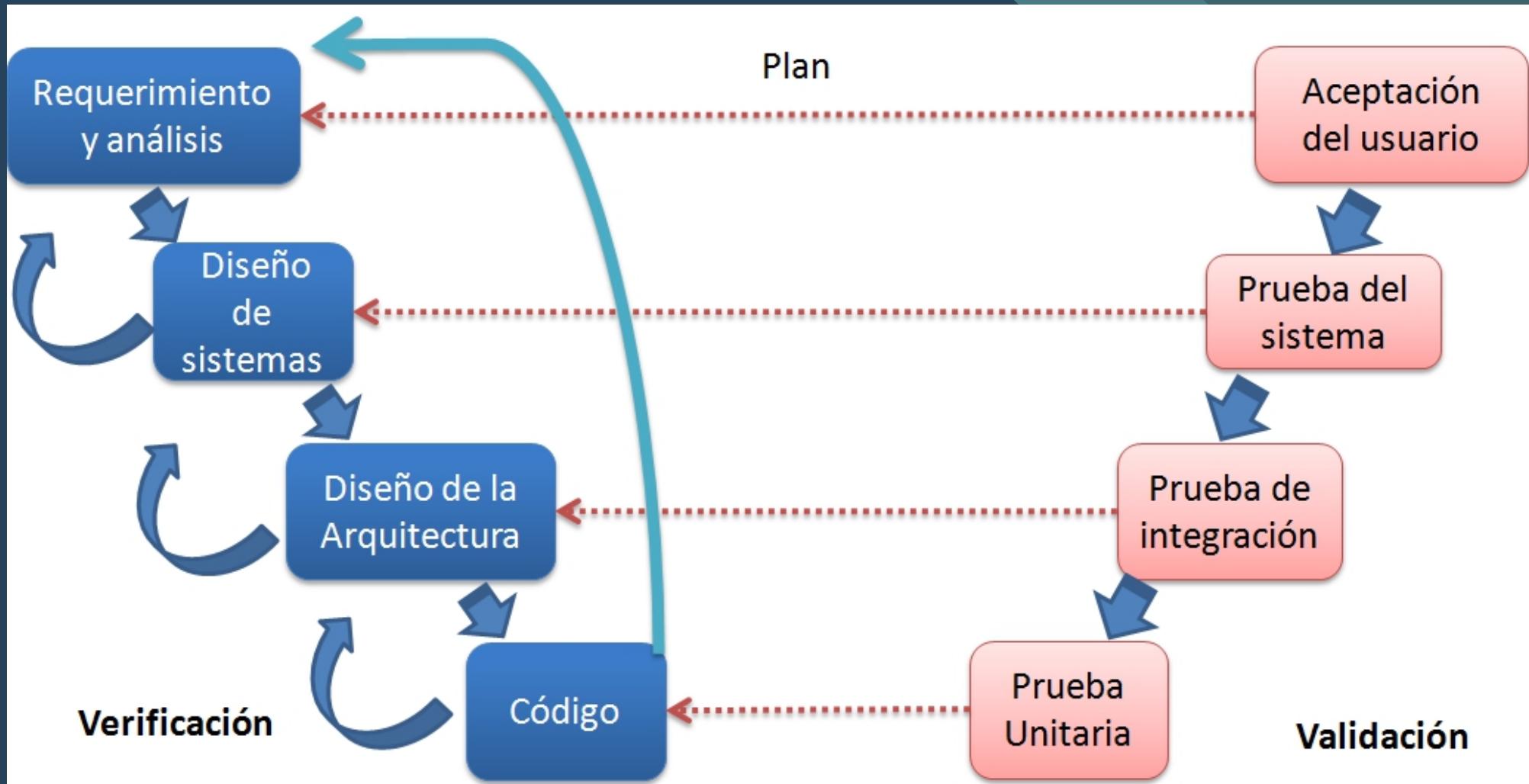
Introducción. Pruebas de Sistema

- Consiste en probar profundamente el sistema para analizar su comportamiento:
 - Pruebas de recuperación: En este tipo de pruebas se refuerza el fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.
 - Prueba de Seguridad. Esta prueba intenta verificar que el sistema está protegido contra accesos ilegales.
 - Prueba de resistencia: Trata de enfrentar el sistema con situaciones que demandan gran cantidad de recursos: memoria, conexiones de red, para ver cómo reacciona (tiempos de respuestas, consumo de recursos, etc.)





Introducción. Modelo en V



Herramientas de Depuración

Investigando nuestro código



Herramientas de Depuración

- Todo entorno de desarrollo, independientemente de la plataforma, así como del lenguaje de programación utilizado, suministra una serie de herramientas de depuración, que nos permiten verificar el código generado.
- Durante el proceso de desarrollo de software, se pueden producir dos tipos de errores: **errores de compilación o errores lógicos**. Cuando ocurre un **error de compilación**, el entorno nos proporciona información de donde se produce y como poder solucionarlo. El programa no puede compilarse hasta que el programador o programadora no corrija ese error.
- El otro tipo de **errores son lógicos**, comúnmente llamados bugs, estos no evitan que el programa se pueda compilar con éxito, ya que no hay errores sintácticos, ni se utilizan variables no declaradas, etc. Sin embargo, los errores lógicos, pueden provocar que el programa devuelva resultados erróneos, que no sean los esperados o pueden provocar que el programa termine antes de tiempo o no termine nunca.
- El depurador permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos. Un programa debe compilarse con éxito para poder utilizarlo en el depurador. El **depurador nos permita analizar todo el programa**, mientras éste se ejecuta. Permite suspender la ejecución de un programa, examinar y establecer los valores de las variables, comprobar los valores devueltos por un determinado método, el resultado de una comparación lógica o relacional, etc.
 - <https://blog.jetbrains.com/idea/2020/05/debugger-basics-in-intellij-idea/>
 - <https://www.jetbrains.com/help/idea/debugging-code.html>



Herramientas de Depuración. Puntos de ruptura

- Dentro del menú de depuración, nos encontramos con la opción **insertar punto de ruptura (breakpoint)**. Se selecciona **la línea de código donde queremos que el programa se pare**, para a partir de ella, inspeccionar variables, o realizar una ejecución paso a paso, para verificar la corrección del código
- Durante la prueba de un programa, puede ser interesante la verificación de determinadas partes del código. No nos interesa probar todo el programa, ya que hemos delimitado el punto concreto donde inspeccionar. Para ello, utilizamos los puntos de ruptura.
- **Los puntos de ruptura son marcadores que pueden establecerse en cualquier línea de código ejecutable** (no sería válido un comentario, o una línea en blanco). Una vez insertado el punto de ruptura, e iniciada la depuración, el programa a evaluar se ejecutaría hasta la línea marcada con el punto de ruptura. **En ese momento, se pueden realizar diferentes labores, por un lado, se pueden examinar las variables, y comprobar que los valores que tienen asignados son correctos, o se pueden iniciar una depuración paso a paso, e ir comprobando el camino que toma el programa a partir del punto de ruptura.** Una vez realiza la comprobación, podemos abortar el programa, o continuar la ejecución normal del mismo.
- Dentro de una aplicación, se pueden insertar varios puntos de ruptura, y se pueden eliminar con la misma facilidad con la que se insertan.



Herramientas de Depuración. Puntos de ruptura

```
1 ► public class AverageFinder {  
2 ►   □ public static void main(String[] args) {  
3 ►     System.out.println("Average finder v0.1");  
4 ●   □   double avg = findAverage(args);  
5 ►     System.out.println("The average is " + avg);  
6 ►   □ }  
7  
8 @ □ private static double findAverage(String[] input) {  
9   □   double result = 0;  
10  □   for (String s : input) {  
11    □     result += Integer.parseInt(s);  
12  □   }  
13  □   return result;  
14  □ }  
15 }
```

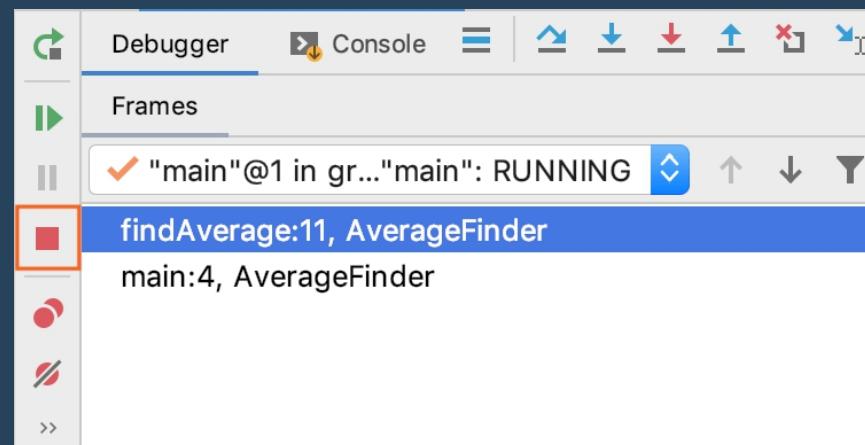
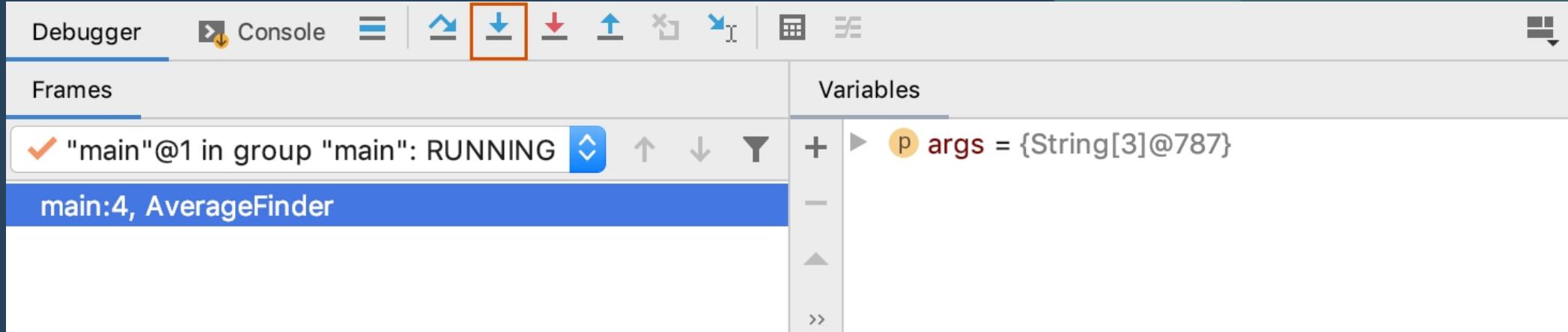


Herramientas de Depuración. Tipos de ejecución

- Para poder depurar un programa, podemos **ejecutar el programa de diferentes formas**, de manera que en función del problema que queramos solucionar, nos resulte más sencillo un método u otro. Nos encontramos con lo siguientes tipo de ejecución: paso a paso por instrucción, paso a paso por procedimiento, ejecución hasta una instrucción, ejecución de un programa hasta el final del programa,
- Algunas veces es necesario **ejecutar un programa línea por línea**, para buscar y corregir errores lógicos. El avance paso a paso a lo largo de una parte del programa puede ayudarnos a verificar que el código de un método se ejecute en forma correcta.
- **El paso a paso por procedimientos**, nos permite introducir los parámetro que queremos a un método o función de nuestro programa, pero en vez de ejecutar instrucción por instrucción ese método, nos devuelve su resultado. Es útil, cuando hemos comprobado que un procedimiento funciona correctamente, y no nos interese volver a depurarlo, sólo nos interesa el valor que devuelve.
- En la **ejecución hasta una instrucción**, el depurador ejecuta el programa, y se detiene en la instrucción donde se encuentra el cursor, a partir de ese punto, podemos hacer una depuración paso a paso o por procedimiento.
- En la **ejecución de un programa hasta el final del programa**, ejecutamos las instrucciones de un programa hasta el final, sin detenernos en las instrucciones intermedias.



Herramientas de Depuración. Tipos de ejecución





Herramientas de Depuración. Examinadores de Variables

- Durante el proceso de implementación y prueba de software, **una de las maneras más comunes de comprobar que la aplicación funciona de manera adecuada, es comprobar que las variables vayan tomando los valores adecuados en cada momento.**
- Los examinadores de variables, forman uno de los elementos más importantes del proceso de depuración de un programa. Iniciado el proceso de depuración, normalmente con la ejecución paso a paso, el programa avanza instrucción por instrucción. Al mismo tiempo, las distintas variables, van tomando diferentes valores. **Con los examinadores de variables, podemos comprobar los distintos valores que adquiere las variables, así como su tipo.** Esta herramienta es de gran utilidad para la detección de errores.



Herramientas de Depuración. Examinadores de Variables

The screenshot shows a Java application running in a debugger. The code in the editor is:

```
617 if (o == null) {
618     for (; i < size; i++)
619         if (es[i] == null)
620             break found;
621     } else {
622         for (; i < size; i++) size: 2
623         if (o.equals(es[i])) o: Point@803 es: Object[10]@804 i: 0
624             break found;
625     }
626     return false;
```

The **Inline Debugger** pane at the top right shows variable values: `size: 2`, `o: Point@803`, `es: Object[10]@804`, and `i: 0`. A green box highlights this area.

The **Variables Pane** at the bottom right lists the current variables:

- `this = {ArrayList@802} size = 2`
- `0 = {Point@806}`
 - `x = 12`
 - `y = 20`
- `1 = {Point@807}`
- `o = {Point@803}`
- `es = {Object[10]@804}`
 - `size = 2`
 - `i = 0`
- `es[i] = {Point@806}`

A green box highlights the `es` variable entry in the `es` list. An orange box highlights the entire **Variables** pane.

The **Frames** pane on the left shows the current stack trace:

- `"main"@1 in group "main": RUNNING`
- `remove:623, ArrayList (java.util)`
- `removeValue:23, Coordinates (com.jetbrains)`
- `main:11, Coordinates (com.jetbrains)`

Excepciones

Manejando errores en tiempo de ejecución



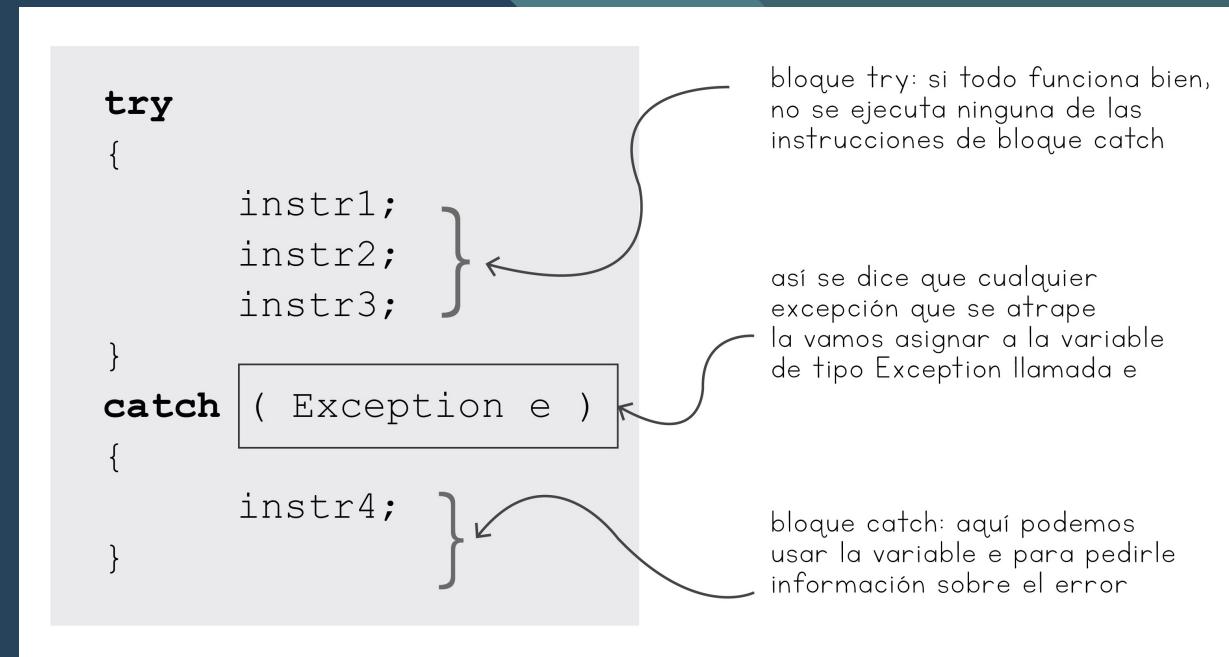
Excepciones

- Una excepción es la indicación de que se produjo un error en el programa la ejecutarse y que de forma inesperada.
- Las excepciones, como su nombre lo indica, se producen cuando la ejecución de un método no termina correctamente, sino que termina de manera excepcional como consecuencia de una situación no esperada.
- Ejemplos: dividir por cero, esperar un entero y recibir un carácter, etc.
- Lo importante es poder evitarlas con antelación. Usando condicionales, mejorando nuestro código.
- Posteriormente, debemos saber reaccionar si aparecen



Excepciones. Try - Catch - Finally

- En la instrucción **try-catch** hay dos obligatorios y uno opcional bloques de instrucciones, con los siguientes objetivos:
- Delimitar la porción de código dentro de un método en el que necesitamos desviar el control si una excepción ocurre allí (la parte try). Si se dispara una excepción en alguna de las instrucciones del bloque try, la ejecución del programa pasa inmediatamente a las instrucciones del bloque catch. Si no se dispara ninguna excepción en las instrucciones del bloque try, la ejecución continúa después del bloque catch.
- Definir el código que manejará el error o atrapará la excepción (la parte catch).
- La parte Finally se ejecutara haya habido error o no, y es opcional.



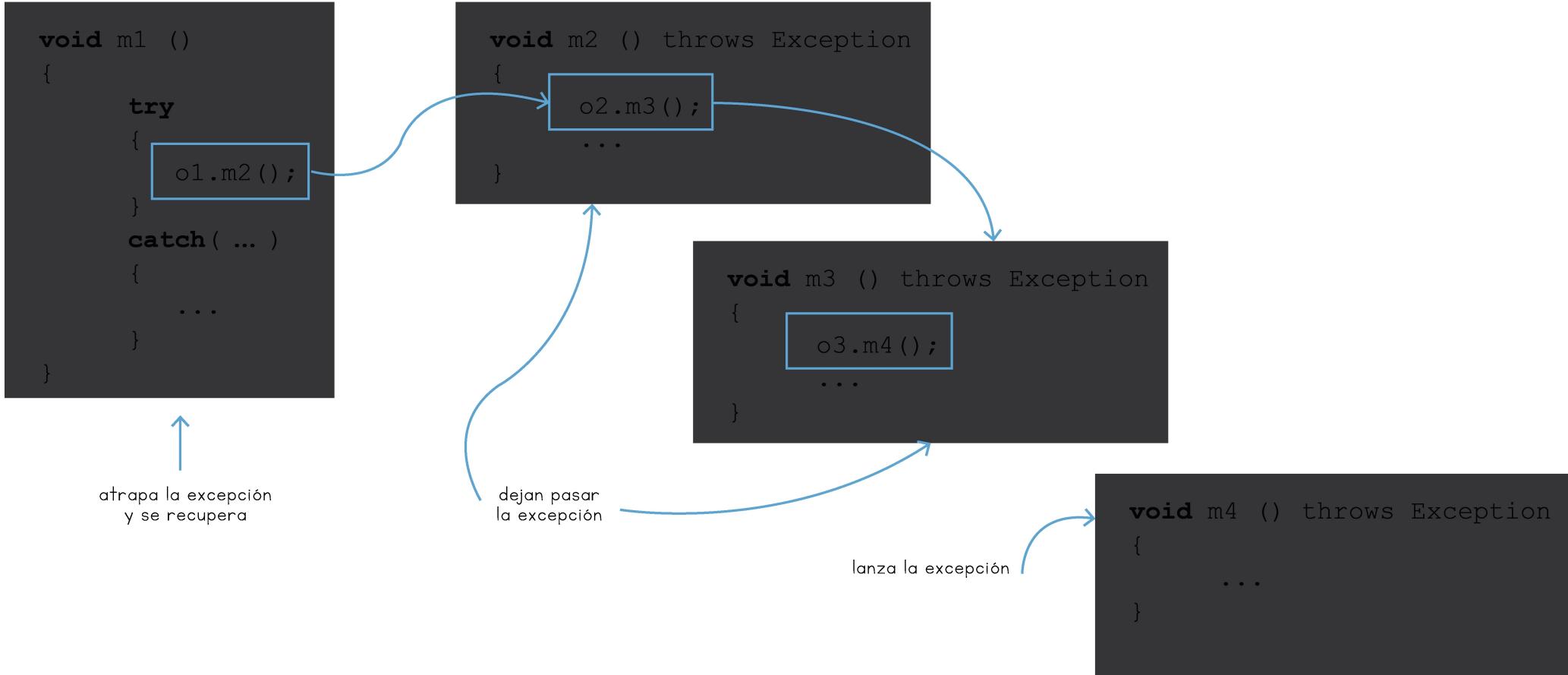


Excepciones. Throw

- Cuando necesitamos disparar una excepción dentro de un método utilizamos la instrucción throw. Esta instrucción recibe como parámetro un Exception, el cual es lanzado o disparado al método que corresponda, siguiendo el esquema planteado anteriormente.
- **Usamos Throw cuando detectamos el error pero no sabemos cómo recuperar el estado del sistema. Entonces será otro método el que usando Try/Catch sepa como actuar y devolver el sistema a un estado estable, ya sea mostrando un mensaje de error, o haciendo los cambios que considere oportunos.**
- **De esta manera conseguimos “atrapar” la excepción.**



Excepciones. Throw



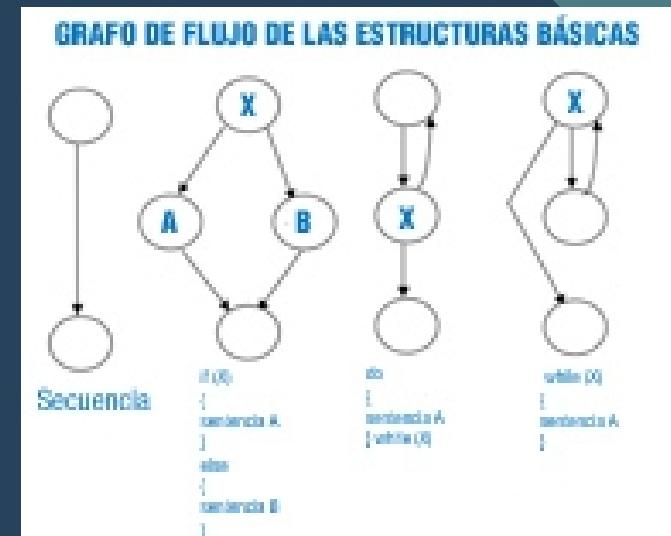
Pruebas de Caja Blanca

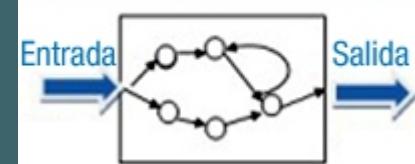
Investigando nuestro código desde dentro



Caja Blanca. Camino básico y Complejidad

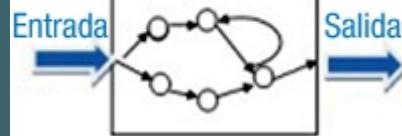
- **Prueba de la Caja Blanca (White Box Testing):** se prueba la aplicación desde dentro, usando su lógica de aplicación.
- Una prueba de Caja Blanca, va a analizar y probar directamente el código de la aplicación. Como se deriva de lo anterior, para llevar a cabo una prueba de Caja Blanca, es necesario un conocimiento específico del código, para poder analizar los resultados de las pruebas.
- **Prueba del camino básico:** La primera es la prueba del camino básico, que se basa en la complejidad del flujo de ejecución desglosado en un conjunto básico de caminos. Los casos en los que se divide la prueba, obtenidos en función del conjunto básico, garantizan que se ejecuta por lo menos una vez cada sentencia del programa. La prueba del camino básico es una técnica de prueba de caja blanca propuesta inicialmente por Tom McCabe .
- Grafo de Control o de Flujo: es El grafico construido a partir de las sentencias de nuestro código.





Caja Blanca. Camino básico y Complejidad

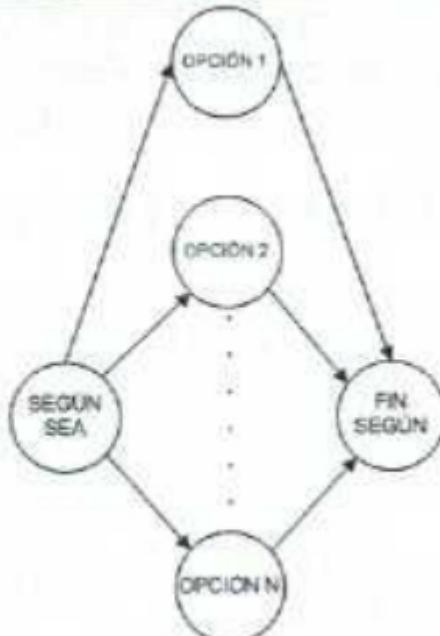
ESTRUCTURA	GRAFO DE FLUJO
SECUENCIAL Instrucción 1 Instrucción 2 Instrucción n	
CONDICIONAL Si <condición> Entonces <Instrucciones> Si no <Instrucciones> Fin si	
HACER MIENTRAS Mientras <condición> Hacer <instrucciones> Fin mientras	
REPETIR HASTA Repetir <instrucciones> Hasta que <condición>	

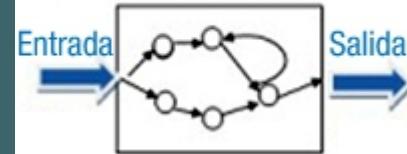


Caja Blanca. Camino básico y Complejidad

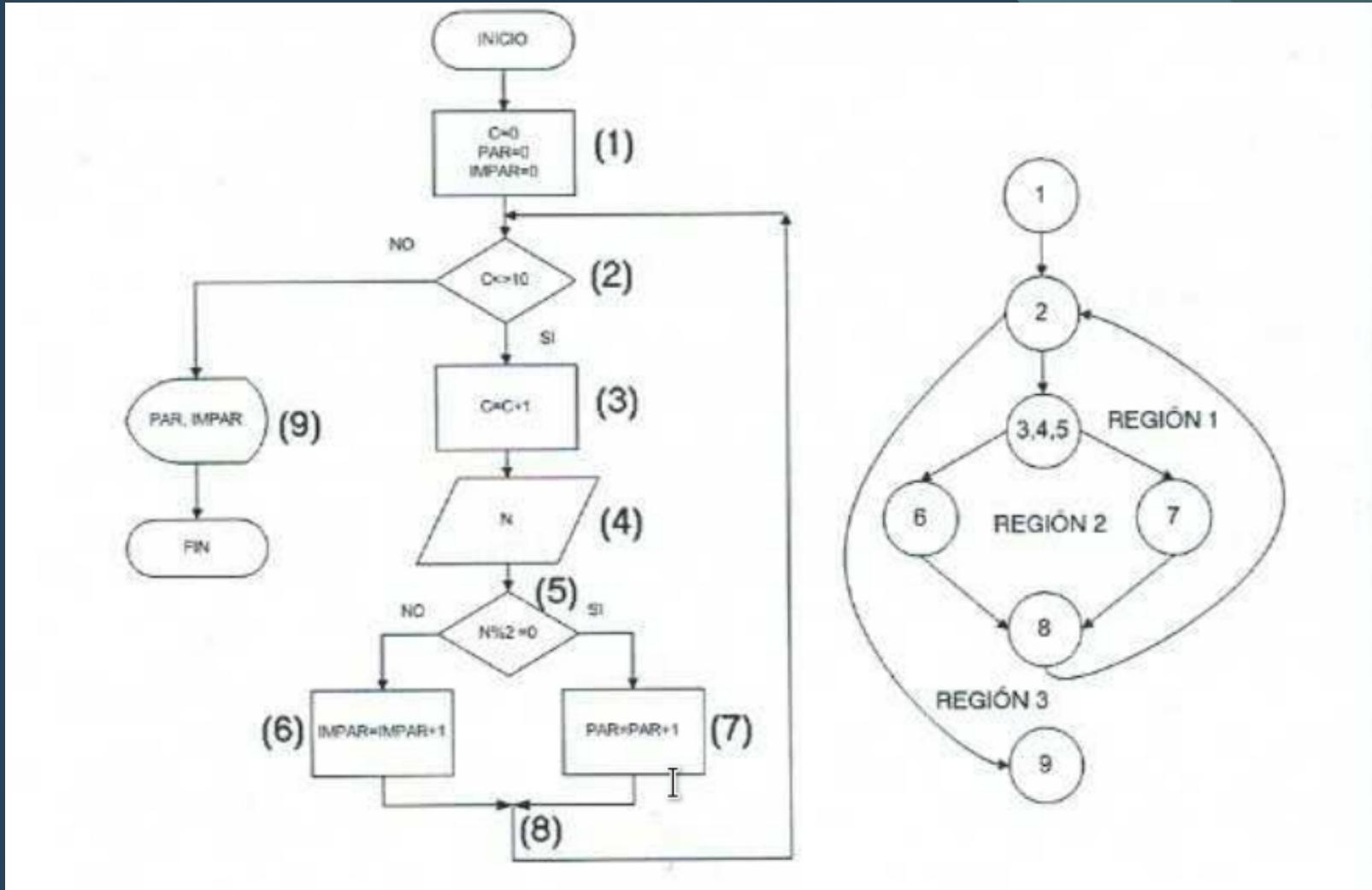
CONDICIONAL MÚLTIPLE

```
Según sea <variable> Hacer
    Caso opción 1:
        <Instrucciones>
    Caso opción 2:
        <Instrucciones>
    Caso opción 3:
        <Instrucciones>
    Otro caso:
        <Instrucciones>
Fin según
```





Caja Blanca. Camino básico y Complejidad

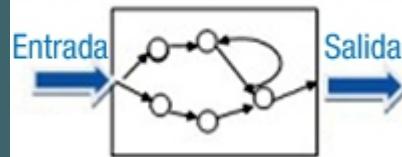




Caja Blanca. Camino básico y Complejidad

- Complejidad Ciclomática $V(G)$: La complejidad ciclomática es un parámetro de los grafos de flujo y representa la complejidad lógica de ejecución del programa. En el contexto de las pruebas, la cuantía de la complejidad ciclomática representa el número de caminos independientes que forman un conjunto de caminos básicos y por ello nos da el número mínimo de pruebas que garantiza la ejecución de cada instrucción al menos una vez. La complejidad ciclomática se calcular de varias formas:
 - $V(G) = R$. Número de regiones en que se subdivide el plano que representa el grafo, considerando la que queda fuera del grafo como una región más.
 - $V(G) = \text{Aristas} - \text{Vértices} + 2$.
 - $V(G) = \text{Nodos Predicado} + 1$.
- Para nuestro ejemplo
 - $V(G) = \text{Número de Regiones} = 3$
 - $V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3$
 - $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$

Complejidad ciclomática	Evaluación de 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.

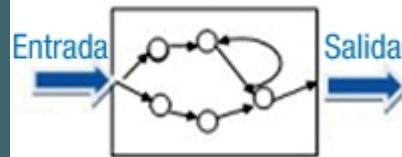


Caja Blanca. Camino básico y Complejidad

- $V(G)$ nos da el número de caminos básicos y con ello la generación de casos de prueba.
- La generación de los casos en los que se divide la prueba se puede realizar siguiendo unos sencillos pasos que se enumeran a continuación:
 1. Partiendo del código fuente se representa el grafo de flujo.
 2. Se determina la complejidad ciclomática.
 3. Se genera un conjunto básico de caminos. O sea, tantos caminos independientes como indica la complejidad ciclomática.
 4. Se establecen los datos que forzarán la ejecución de cada camino del conjunto básico.

- De nuestro ejemplo podemos tener 3 caminos.
 - C1: 1 - 2 - 9
 - C2: 1 - 2 - 3 - 4 - 5 - 6 - 8 - 2 - 9
 - C3: 1 - 2 - 3 - 4 - 5 - 7 - 8 - 2 - 9

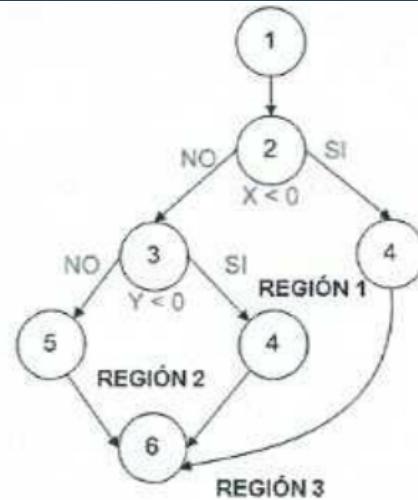
Camino	Caso de prueba	Resultado esperado
1	Escoger algún valor de C tal que NO se cumpla la condición $C > 10$ $C=10$	Visualizar el número de pares y el de impares
2	Escoger algún valor de C tal que Sí se cumpla la condición $C > 10$. Escoger algún valor de N tal que NO se cumpla la condición $N \% 2 = 0$ $C= 1, N = 5$	Contar números impares
3	Escoger algún valor de C tal que Sí se cumpla la condición $C > 10$. Escoger algún valor de N tal que Sí se cumpla la condición $N \% 2 = 0$ $C= 2, N = 4$	Contar números pares



Caja Blanca. Camino básico y Complejidad

```

static void visualizarMedia(float x, float y) {
    float resultado = 0; (1)
    if( x < 0 || y < 0) (3)
        (2) System.out.println("X e Y deben ser positivos"); (4)
    else {
        resultado = (x + y) / 2;
        System.out.println("La media es: " + resultado);
    } (6)
}
  
```



$$V(G) = \text{Regiones} = 3$$

$$V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3$$

$$V(G) = \text{Predicados} + 1 = 2 + 1 = 3$$

Camino	Caso de prueba	Resultado esperado
Camino 1: 1-2-3-5-6	Escoger algún X e Y tal que NO se cumpla la condición $X < 0 \ \ Y < 0$ $X=4, Y=5$ visualizarMedia(4,5)	Visualiza: La media es: 4.5
Camino 2: 1-2-4-6	Escoger algún X tal que Sí se cumpla la condición $X < 0$ (Y puede ser cualquier valor) $X=-4, Y=5$ visualizarMedia(-4,5)	Visualiza: X e Y deben ser positivos
Camino 3: 1-2-3-4-6	Escoger algún X tal que NO cumpla la condición $X < 0$ y escoger algún Y que Sí cumpla la condición $Y < 0$ $X=4, Y=-5$ visualizarMedia(4,-5)	Visualiza: X e Y deben ser positivos



Caja Blanca. Camino básico y Complejidad

- **Cobertura de Sentencias:** Se trata de ejecutar con los casos de prueba cada sentencia e instrucción al menos una vez. Ejecutando los casos de prueba que nos dan en el enunciado ejecutaremos cada instrucción al menos una vez: si nos fijamos en los caminos independientes nos daremos cuenta de que el flujo de ejecución pasa por todos los nodos del código.
- **Cobertura de Decisiones:** Escribimos los casos suficientes para que cada condición tenga al menos una resultado verdadero y otro falso. En nuestro caso bastaría con ejecutar el interior del if y el interior del else, sin tener en cuenta las condiciones. Cuando tenemos decisiones multicondicionales puede ejecutarse el bloque else con diferentes combinaciones estas.
- **Cobertura de Condiciones:** Se trata de escribir los casos suficientes para que cada condición de cada decisión adopte el valor verdadero y el falso al menos una vez. Los casos de prueba en este caso serán los mismos que en la cobertura de decisiones.
- **Cobertura de Decisión/Condición:** Es el cumplimiento de la cobertura de condiciones y de decisiones.
- **Cobertura de Condición Múltiple:** Si tenemos decisiones multicondicionales las descompondremos en decisiones unicondicionales, ejecutando todas las combinaciones posibles de resultados.

Pruebas de Caja Negra

Investigando nuestro código desde fuera



Caja Negra



- **Prueba de la Caja Negra (Black Box Testing):** cuando una aplicación es probada usando su interfaz externa, sin preocuparnos de la implementación de la misma. Aquí lo fundamental es comprobar que los resultados de la ejecución de la aplicación, son los esperados, en función de las entradas que recibe.
- Una prueba de tipo Caja Negra se lleva a cabo sin tener que conocer ni la estructura, ni el funcionamiento interno del sistema. Cuando se realiza este tipo de pruebas, solo se conocen las entradas adecuadas que deberá recibir la aplicación, así como las salidas que les correspondan, pero no se conoce el proceso mediante el cual la aplicación obtiene esos resultados.
- Nuestras pruebas son:
 - Particiones de equivalencia
 - Prueba de Valores Límite



Caja Negra. Clases de Equivalencia



- Las clases de equivalencia, es un tipo de prueba funcional, en donde cada caso de prueba, pretende cubrir el mayor número de entradas posible.
- El dominio de valores de entrada, se divide en número finito de clases de equivalencia. Como la entrada está dividida en un conjunto de clases de equivalencia, la prueba de un valor representativo de cada clase, permite suponer que el resultado que se obtiene con él, será el mismo que con cualquier otro valor de la clase.
- Cada clase de equivalencia debe cumplir:
 - Si un parámetro de entrada debe estar comprendido entre un determinado rango, hay tres clases de equivalencia: por debajo, en y por encima.
 - Si un parámetro de entrada, requiere un valor específico, se define una clase de equivalencia y dos no válidas.
 - Si una entrada requiere un valor entre los de un conjunto, aparecen dos clases de equivalencia: en el conjunto o fuera de él.
 - Si una entrada es booleana, hay dos clases: sí o no.
- Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases. Tanto para las clases válidas como las clases inválidas.



Caja Negra. Clases de Equivalencia



Condiciones de entrada	Nº de Clases de equivalencia válidas	Nº de Clases de equivalencia no válidas
1 . Rango	I 1 CLASE VÁLIDA Contempla los valores del rango	2 CLASES NO VÁLIDAS Un valor por encima del rango Un valor por debajo del rango
2. Valor específico	1 CLASE VÁLIDA Contempla dicho valor	2 CLASES NO VÁLIDAS Un valor por encima Un valor por debajo
3. Miembro de un conjunto	1 CLASE VÁLIDA Una clase por cada uno de los miembros del conjunto	1 CLASE NO VÁLIDA Un valor que no pertenece al conjunto
4. Lógica	1 CLASE VÁLIDA Una clase que cumpla la condición	1 CLASE NO VÁLIDA Una clase que no cumpla la condición



Caja Negra. Clases de Equivalencia



Clases de Equivalencia:

Por debajo: $x < 0$

En: $x > 0$ y $x < 100$

Por encima: $x > 100$

y los respectivos casos de prueba, podrían ser:

Por debajo: $x = 0$

En: $x = 50$

Por encima: $x = 100$

```
public double funcion1 (double x)
{
    if (x > 0 && x < 100)
        return x+2;
    else
        return x-2;
}
```



Caja Negra. Clases de Equivalencia



Numero-empleado es un campo de números enteros positivos de 3 dígitos (excluido el 000).

Nombre-empleado es un campo alfanumérico de 10 caracteres.

Meses-Trabajo es un campo que indica el número de meses que lleva trabajando el empleado; es un entero positivo (incluye el 000) de 3 dígitos.

Directivo es un campo de un solo carácter que puede ser «+» para indicar que el empleado es un directivo y «-» para indicar que no lo es.



Caja Negra. Clases de Equivalencia



Número-empleado es un campo de números enteros positivos de 3 dígitos (excluido el 000).

Nombre-empleado es un campo alfanumérico de 10 caracteres.

Meses-Trabajo es un campo que indica el número de meses que lleva trabajando el empleado; es un entero positivo (incluye el 000) de 3 dígitos.

Directivo es un campo de un solo carácter que puede ser «+» para indicar que el empleado es un directivo y «-» para indicar que no lo es.

Condición	Clases Válidas	Clases Inválidas
Nº empleado	1. Número de 3 dígitos mayor a 000 y menor o igual a 999	2. Número menor a 3 dígitos 3. Número mayor a 3 dígitos 4. Número 000 5. Número negativo 6. No es número 7. Cadena Nula
Nombre empleado	8. Cadena alfanumérica de 10 caracteres.	9. Cadena de más de 10 caracteres. 10. Cadena de menos de 10 caracteres. 11. Cadena sólo de dígitos 12(7).
Meses trabajados	13(1,4).	14(2). 15(3). 16(5). 17(6). 18(7).
Directivo	19. Cadena de 1 carácter = '+' 20. Cadena de 1 carácter = '-'	21. Cadena de 1 carácter distinto a '+' o '-' 22. Cadena de más de 1 carácter 23(7).

I	Caso de Prueba	Clases Válidas	Clases Invalidas	Salida
	(625, JORGE_SOTO, 035, '+')	1, 8, 13, 19	---	P1
	(021, JUAN_PEREZ, 012, '-')	1, 8, 13, 20	---	P2
	(125, MARIA_LASO, 010, '+')	1, 8, 13, 19	---	P3
	(003, ANA_ROBLES, 005, '-')	1, 8, 13, 20	---	P4
	(45, BART_SIMPSON, 15, '=')	---	2, 9, 14, 21	Error
	(0075, CARTMAN, 1020, '+-')	---	3,10,15,22	Error
	(000, 0023456789, -03, null)	---	4,11, 16, 23	Error
	(-89, null, dos, '-')	20	5, 12, 17	Error
	(olo, BETO_SANTO, null, '+')	8, 19	6, 18	Error
	(null, TITO_VILLA, 018, '-')	8, 13, 20	7	Error



Caja Negra. Valores Límite



- La prueba de Valores Límite nos dice que los errores tienden a producirse con más probabilidad en los límites de las entradas.
- La experiencia ha demostrado que los casos de prueba que obtienen una mayor probabilidad de éxito, son aquellos que trabajan con valores límite.
- Esta técnica, se suele utilizar como complementaria de las particiones equivalentes, pero se diferencia, en que se suelen seleccionar, no un conjunto de valores, sino unos pocos, en el límite del rango de valores aceptado por el componente a probar.
- Cuando hay que seleccionar una valor para realizar una prueba, se escoge aquellos que están situados justo en el límite de los valores admitidos.

	Condiciones de entrada y salida	Casos de prueba
Código	Entero de 1 a 100	Valores: 0, 1, 100, 101
Puesto	Alfanumérico de hasta 4 caracteres	Longitud de caracteres: 0, 1, 4, 5
Antigüedad	De 0 a 25 años (Real)	Valores: 0, 25, -0.1, 25.1
Horas semanales	De 0 a 60	Valores: 0, 60, -1, 61
Fichero de entrada	Tiene de 1 a 100 registros	Para leer 0, 1, 100 y 101 registros
Fichero de salida	Podrá tener de 0 a 10 registros	Para generar 0, 10 y 11 registros (no se puede generar -1 registro)
Array interno	De 20 cadenas de caracteres	Para el primer y último elemento.



Caja Negra. Clases de Equivalencia



Número-empleado es un campo de números enteros positivos de 3 dígitos (excluido el 000).

Nombre-empleado es un campo alfanumérico de 10 caracteres.

Meses-Trabajo es un campo que indica el número de meses que lleva trabajando el empleado; es un entero positivo (incluye el 000) de 3 dígitos.

Directivo es un campo de un solo carácter que puede ser «+» para indicar que el empleado es un directivo y «-» para indicar que no lo es.

I	Caso de Prueba	Clases Válidas	Clases Invalidas	Salida
	(625, JORGE_SOTO, 035, '+')	1, 8, 13, 19	---	P1
	(021, JUAN_PEREZ, 012, '-')	1, 8, 13, 20	---	P2
	(125, MARIA_LASO, 010, '+')	1, 8, 13, 19	---	P3
	(003, ANA_ROBLES, 005, '-')	1, 8, 13, 20	---	P4
	(45, BART_SIMPSON, 15, '=')	---	2, 9, 14, 21	Error
	(0075, CARTMAN, 1020, '+-')	---	3,10,15,22	Error
	(000, 0023456789, -03, null)	---	4,11, 16, 23	Error
	(-89, null, dos, '-')	20	5, 12, 17	Error
	(olo, BETO_SANTO, null, '+')	8, 19	6, 18	Error
	(null, TITO_VILLA, 018, '-')	8, 13, 20	7	Error



Caja Negra. Clases de Equivalencia



Número-empleado es un campo de números enteros positivos de 3 dígitos (excluido el 000).

Nombre-empleado es un campo alfanumérico de 10 caracteres.

Meses-Trabajo es un campo que indica el número de meses que lleva trabajando el empleado; es un entero positivo (incluye el 000) de 3 dígitos.

Directivo es un campo de un solo carácter que puede ser «+» para indicar que el empleado es un directivo y «-» para indicar que no lo es.

Condición	Clases Válidas	Clases Inválidas
Nº empleado	24. Número 001 25. Número 002 26. Número 999 27. Número 998	28 (4) 29. Numero 1000 30. Número de 2 dígitos 31. Numero de 4 dígitos
Nombre empleado	32(8).	33. Cadena de 11 caracteres. 34. Cadena de 9 caracteres.
Meses trabajados	35(4). 36(24). 37(26). 38(27).	39. Número -01 40(29). 41(30). 42(31).
Directivo	43(19). 44(20).	45. Cadena de 2 caracteres

Caso de Prueba	Clases Válidas	Clases Invalidas	Salida
(001, JORGE_SOTO, 000, '+')	24, 32, 35, 43	---	P3
(002, JUAN_PEREZ, 001, '-')	25, 32, 36, 44	---	P4
(999, MARIA_LASO, 999, '+')	26, 32, 37, 43	---	P1
(998, ANA_ROBLES, 998, '-')	27, 32, 38, 44	---	P2
(000, MONTY_BURNS, -01, '++')	---	28, 33, 39, 45	Error
(1000, STANSMITH, 1000, '+')	43	29, 34, 40	Error
(89, JUAN_SANTO, 78, '-')	32, 44	30, 41	Error
(0233, ANA_CARASO, 0190, '+')	32, 43	31, 42	Error

“

“Por norma, los sistemas software no
funcionan bien hasta que han sido
utilizados y han fallado repetidamente
en entornos reales”

-- Dave Parnas

”

Principios del TDD

Aplicando Test-Driven Development



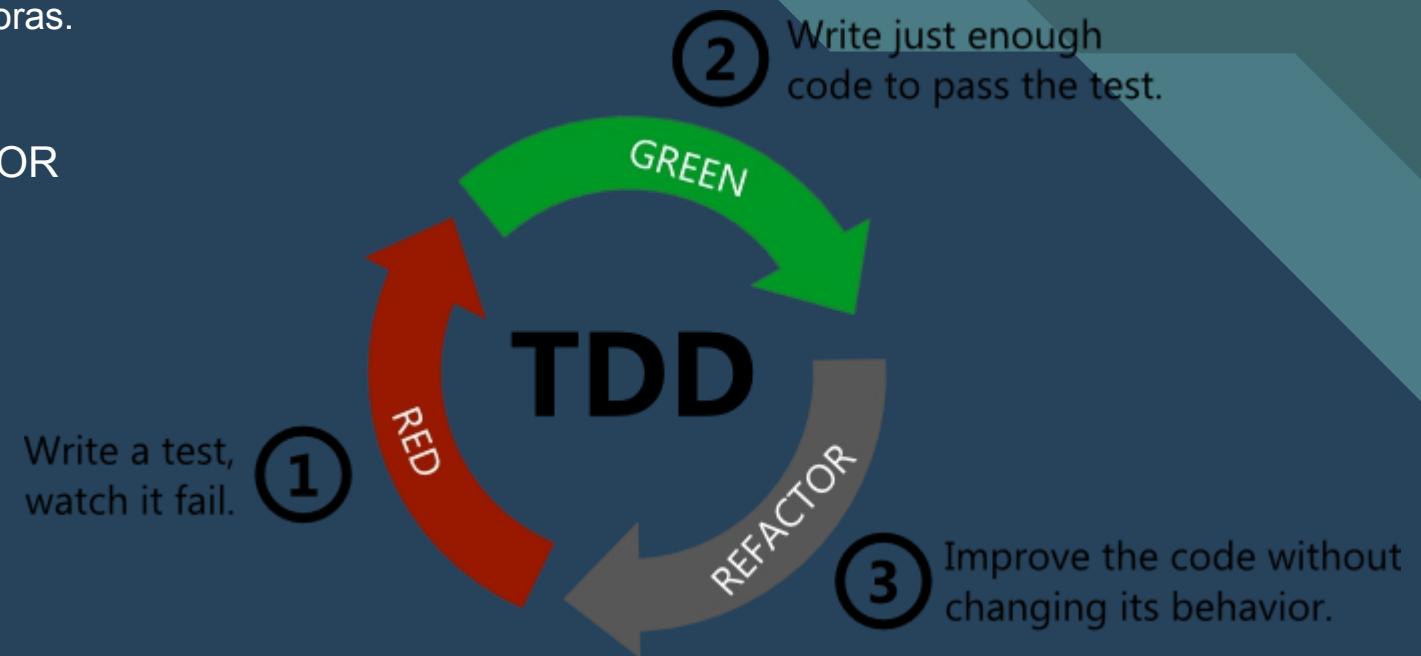
TDD. Introducción

- El Desarrollo Dirigido por Tests (Test Driven Development), al cual me referiré como TDD, es una técnica de diseño e implementación de software incluida dentro de la metodología XP. Coincido con Peter Provost en que el nombre es un tanto desafortunado; algo como Diseño Dirigido por Ejemplos hubiese sido quizás mas apropiado. TDD es una técnica para diseñar software que se centra en tres pilares fundamentales:
 - La implementación de las funciones justas que el cliente necesita y no más.
 - La minimización del número de defectos que llegan al software en fase de producción.
 - La producción de software modular, altamente reutilizable y preparado para el cambio.
- Ofrece las ventajas:
 - Incrementa la productividad.
 - Nos hace descubrir y afrontar más casos de uso en tiempo de diseño.
 - La jornada se hace mucho más amena.
 - Uno se marcha a casa con la reconfortante sensación de que el trabajo está bien hecho.



TDD. Algoritmo de TDD

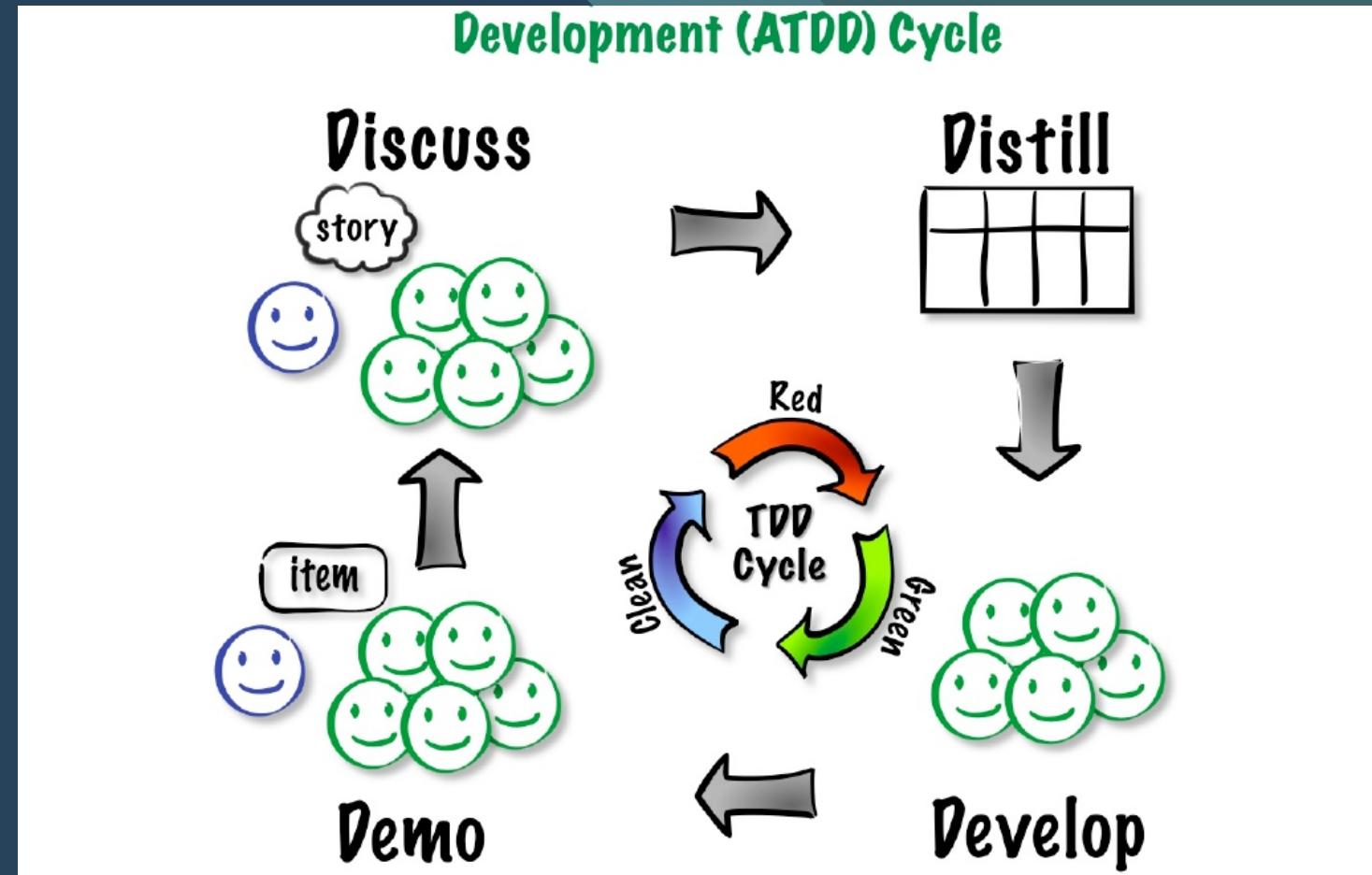
- La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD sólo tiene tres pasos:
 - Escribir la especificación del requisito (el ejemplo, el test).
 - Implementar el código según dicho ejemplo.
 - Refactorizar para eliminar duplicidad y hacer mejoras.
- O lo que es lo mismo, RED - GREEN - REFACTOR
 - Crea el test
 - Comprueba que falla (pos no tienes código aún)
 - Escribe el código justo para pasarlo
 - Comprueba que funciona
 - Refactoriza
 - Vuelve al primer paso
 - Y disfruta :)





TDD. ADD

- Los tests de aceptación o de cliente son el criterio escrito de que el software cumple los requisitos de negocio que el cliente demanda.
- Los requisitos se traducen por ejemplos ejecutables (de como se ejecuta una funcionalidad con sus entradas y salidas esperadas) surgidos del consenso entre los distintos miembros del equipo, incluido por supuesto el cliente.
- Una vez que tenemos los ATDD, se crea el test que lo representa, y posteriormente iniciamos TDD, de esta manera el código que pasa el test se asegura que cumple con el requisito a conseguir.



JUnit

Test Unitarios en JAVA



JUnit

- Cuando se implementa software, resulta recomendable comprobar que el código que hemos escrito funciona correctamente. Para ello, implementamos pruebas que verifican que nuestro programa genera los resultados que de él esperamos.
- Su vez este conjunto de pruebas nos ayuda a limpiar nuestro código, aplicando TDD y Refactorizando y mejorando el código desarrollado en cada paso.
- JUnit es una suite para automatizar pruebas unitarias basadas en la filosofía AAA (Arrange (Preparar), Act (Actuar), Assert (Afirmar)).
- ¿Qué nos ofrece JUnit?
 - JUnit permite mantener de forma separada los casos de prueba
 - Permite ejecutarlos (y re-ejecutarlos) de forma automática
 - Nos permite construir "árboles de casos de prueba" (suites)
- Las pruebas se hacen sobre clases o métodos aislados para probar su comportamiento en base a resultados esperados que conocemos (TDD).

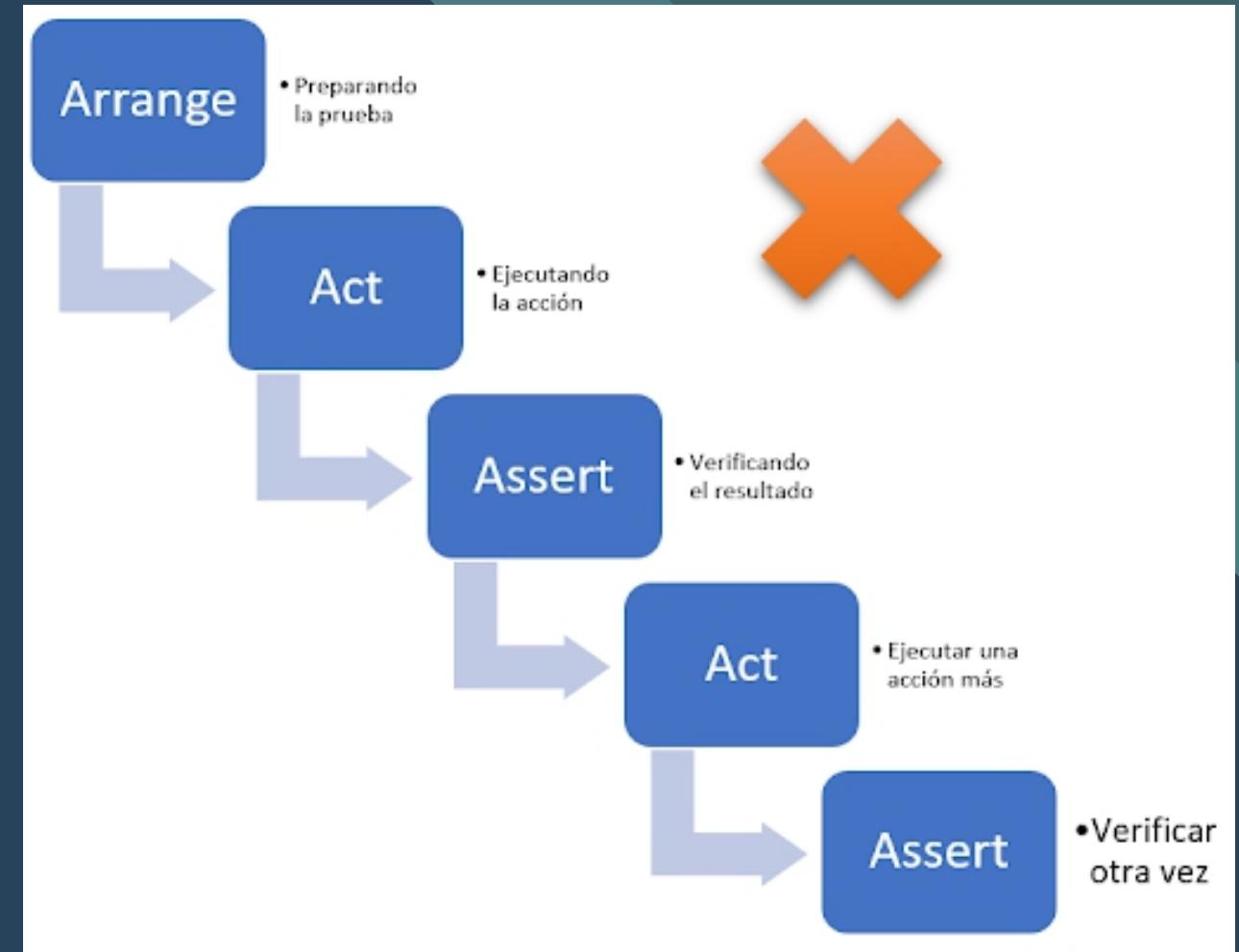


JUnit. Metodología

- JUnit es una suite para automatizar pruebas unitarias basadas en la filosofía AAA (Arrange (Preparar), Act (Actuar), Assert (Afirmar)).
- Para ello preparamos la prueba y los datos básicos para que funcione (Arrange). Es decir, los datos actuales y los datos esperados basados en algunas de las pruebas que hemos visto anteriormente (Caja Blanca o Caja Negra)
- Actuamos o llamamos al método que queremos probar (Act)
- Comprobamos si el resultado obtenido es igual al resultado esperado para el caso de prueba siguiente nuestro diseño de la misma (Assert).

Preparar:

año 200





JUnit. Metodología

Test

Preparar:

año 200

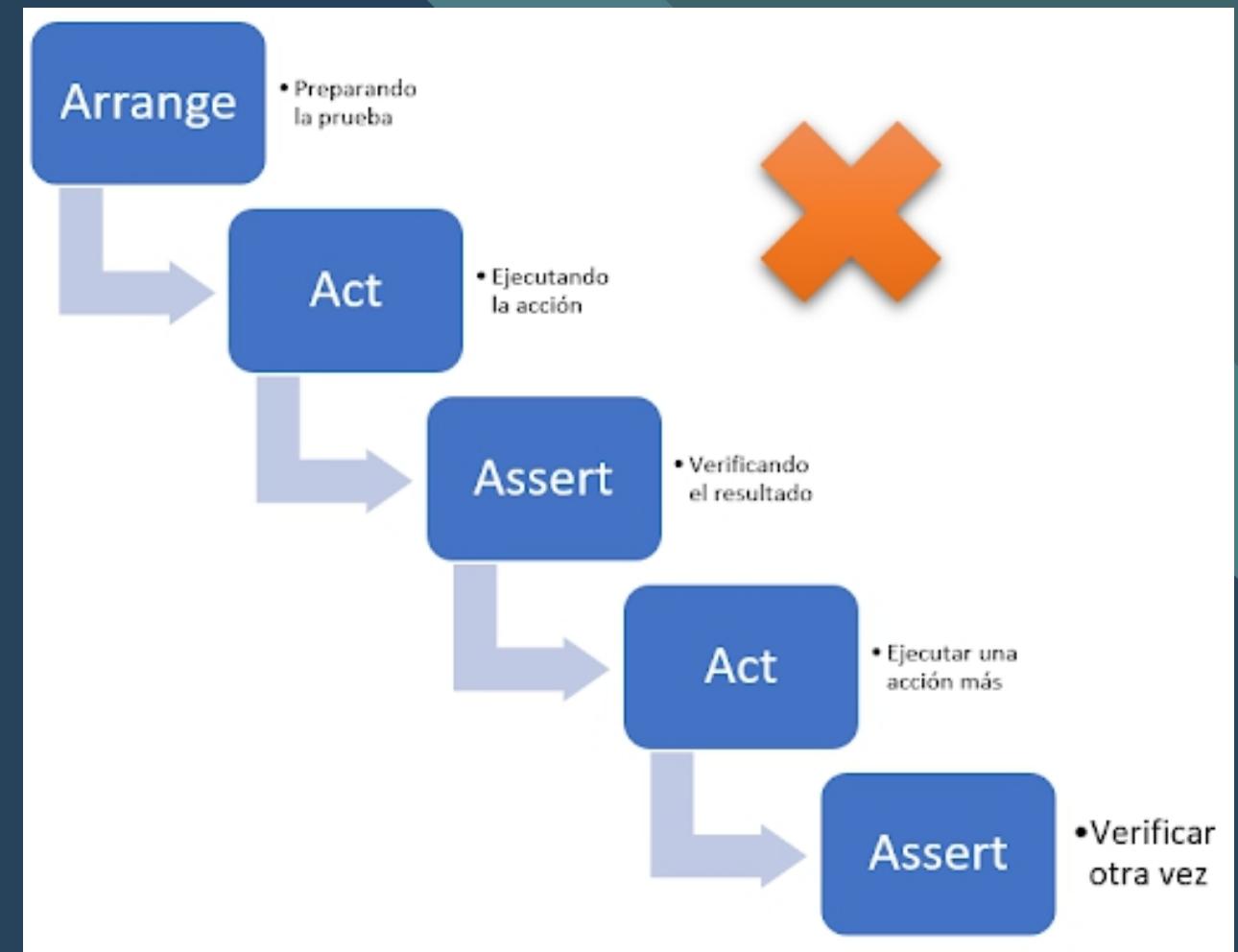
esBisiestoEsperado = true

Actuar:

```
esBisiesto = esBiestoAño(2000);
```

Afirmar:

```
asserEquals(esBisiesto, esBisiestoEsperado);
```





JUnit. Asercciones

- **assertEquals** y **asserNotEquals**: compara valores simples
- **assertArrayEquals**: compara dos arrays
- **assertNotNull** y **assertNull**: compara si es Null o not Null
- **assertNotSame** y **assertSame**: compara dos objetos si son el mismo
- **assertTrue** y **assertFalse**: compara si es verdadero o falso
- fail: compara si lanza una excepción o falla
- assertThat: comapra en función a un Matcher
- assertAll: lanza varias asercciones y es verdadero si se cumplen todas
- assertIterableEquals: compara dos iterables
- assertLinesMatch: compara si una linea coincide con un patrón dado
- **assertTimeout** y **assertTimeoutPreemptively**: Compara sobre un límite de tiempo dado.



JUnit. Asercciones

Método	Parámetros	Significado
assertEquals	(double expected, double actual, double delta)	Indica si dos valores <i>double</i> son iguales dentro de un determinado delta
	(float expected, float actual, float delta)	Indica si dos valores <i>float</i> son iguales dentro de un determinado delta
	(Object[] expecteds, Object[] actuals)	Indica si dos arrays de objetos son iguales
	(Object expected, Object actual)	Indica si dos objetos son iguales
assertFalse	(boolean condition)	Comprueba que la condición pasada es falsa
assertNotNull	(Object object)	Comprueba que un objeto NO es nulo
assertNotSame	(Object unexpected, Object actual)	Comprueba que dos objetos NO se están refiriendo al mismo objeto (comparación de identidad en vez de igualdad)
assertNull	(Object object)	Comprueba que un objeto es nulo
assertSame	(Object unexpected, Object actual)	Comprueba que dos objetos se están refiriendo al mismo objeto (comparación de identidad en vez de igualdad)
assertTrue	(boolean condition)	Comprueba que la condición pasada es cierta
fail	()	Provoca que el test falle

Nota: todos los métodos tienen una segunda versión que acepta como primer parámetro un String en el que mandar un mensaje



JUnit. Anotaciones

- **@Test:** Indica que ese método ejecuta un Test
- **@Before:** El método anotado de esta manera se ejecutará antes de cada test. Se suele usar para iniciar datos.
- **@After:** El método anotado de esta manera se ejecutará después de cada test. Se suele usar para limpiar datos.
- **@BeforeClass / BeforeAll:** Sólo puede haber un método con esta etiqueta. Este método es invocado al principio del lanzamiento de todas las pruebas. Se usa para iniciar datos comunes a todas las pruebas.
- **@AfterClass / AfterAll:** Sólo puede haber un método con esta etiqueta. Este método es invocado al una vez finalizadas todas las pruebas. Se usa para limpiar los datos de todas las pruebas.



JUnit. Anotaciones

Anotación	Parámetros	Significado
@Test	...	Indica a JUnit que el método <i>public void</i> que está siendo anotado puede ser ejecutado como un test. Todas las excepciones lanzadas por el método significarán un fallo en el test. Si no se lanza ninguna excepción se entenderá que el test terminó exitosamente
	expected	<i>@Test(expected = miClaseException.class)</i> indica que se espera que el test lance una excepción, si dicha excepción no se lanza (o se lanza una distinta) el test falla
	timeout	<i>@Test(timeout=100)</i> indica que si el tiempo de ejecución del método es mayor que la cantidad especificada en milisegundos el método falla
@BeforeClass		Indica que un método, que debe ser <i>public static void</i> y no tener argumentos, se ejecute una vez antes que cualquiera de los métodos de test de la clase
@Before		Indica que un método, que debe ser <i>public void</i> , se ejecute antes que cualquiera de los métodos de test de la clase. Se utiliza generalmente para crear instancias que posteriormente comparten todos los tests
@AfterClass		Indica que un método, que debe ser <i>public static void</i> , se ejecute una vez después de la ejecución de todos los métodos de test de la clase. El método <i>AfterClass</i> se ejecuta siempre incluso aunque el método <i>BeforeClass</i> haya terminado en una excepción
@After		Indica que un método, que debe ser <i>public void</i> , se ejecute después de cualquiera de los métodos de test de la clase. Se garantiza que este método se ejecuta incluso aunque el método <i>Before</i> o el método <i>Test</i> terminen en una excepción
@Ignore	...	Permite deshabilitar temporalmente un test
	value	<i>@Ignore("not ready yet") @Test public void something() { ... }</i> indica el motivo



JUnit. Cobertura

- ¿Qué es la cobertura?
 - La cobertura es la cantidad de código (medida porcentualmente) que está siendo cubierto por las pruebas. O sea, ejecuto las pruebas de mi aplicación y si hay alguna línea de mi código que nunca fue ejecutada en el contexto de las pruebas, entonces dicha línea no está cubierta. Si mi código consta de 100 líneas y solo 50 líneas están siendo ejecutadas al correr las pruebas, entonces mi cobertura es del 50%. La pregunta que viene a continuación es:
- ¿Qué beneficio tiene medir la cobertura? y más específicamente ¿qué beneficios tiene tener una alta cobertura?
 - Una respuesta genérica podría ser que aumenta la calidad de mi aplicación. Siendo más concreto podría decir que si tengo una alta cobertura, significa que gran parte de mi código está siendo probado y por consiguiente podría tener cierta certeza sobre el correcto funcionamiento de mi aplicación. Al mismo tiempo medir la cobertura podría ayudarme a detectar código innecesario en mi aplicación, ya que es código que no se ejecuta.
- ¿Qué porcentaje de cobertura es suficiente?
 - La respuesta no es única, existen distintos criterios y pueden resultar bastante polémicos lo ideal es de al menos 90% y en algunos el 80%.
- ¿Una cobertura del 100% asegura que mi código no tiene bugs?
 - De ninguna manera, una cobertura del 100% solo nos dice que todo nuestro código está siendo cubierto por pruebas, pero puede que las pruebas no estén contemplando algunas situaciones, o sea, que faltan pruebas o incluso podría ocurrir que las pruebas fueran deficientes.



JUnit. IntelliJ

- Lo mejor es usar un proyecto Maven o agregar JUnit a tu proyecto y lo vamos a ver en clase
- <https://www.jetbrains.com/help/idea/testing.html>
- <https://blog.jetbrains.com/idea/2020/09/writing-tests-with-junit-5/>
- <https://www.jetbrains.com/help/idea/junit.html>



Recursos

- Twitter: <https://twitter.com/joseluisgonsan>
- GitHub: <https://github.com/joseluisgs>
- Web: <https://joseluisgs.github.io>
- Discord: <https://discord.gg/WKKvSJCS>
- Aula Virtual: <https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=246>



Gracias

José Luis González Sánchez

