Diseño y realización de pruebas

1. INTRODUCCIÓN

- Las pruebas son necesarias en la fabricación de cualquier producto industrial y, de forma análoga, en el desarrollo de proyectos informáticos.
- Una aplicación informática no puede llegar a las manos del usuario con errores, y menos si estos son suficientemente visibles y claros como para haber sido detectados por desarrolladores. Esto provoca una sensación de falta de profesionalidad y disminuye la confianza por parte de los usuarios, que podría mermar oportunidades futuras.
- ¿Cuándo hay que llevar a cabo las pruebas? ¿Qué hay que probar? Constituyen una de las etapas del desarrollo de software y se integran dentro de las diferentes fases del ciclo de vida del software.
- También son muy importantes las pruebas que se llevan una vez finalizado el proyecto. La fase de pruebas del desarrollo de un proyecto es básica.

- Cualquier miembro del equipo de trabajo de un proyecto informático puede cometer errores y estos se pueden dar en cualquiera de las fases del proyecto: análisis, diseño o codificación entre otros.
- Un error no detectado al inicio del desarrollo de un proyecto puede llegar a necesitar cincuenta veces más esfuerzos para ser solucionado que si es detectado a tiempo.
- En un proyecto de desarrollo de software se puede dedicar incluso del 30-50% del coste a la fase de pruebas. Con esto podemos ver su importancia.
- Evaluaremos la calidad del software desarrollado durante todo el ciclo de vida, validando que hace lo que tiene que hacer y que lo hace tal y como se diseñó, a partir de los requierimientos.

2. TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

- Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollados para conseguir un objetivo particular o condición de prueba.
- Para llevar a cabo un caso de prueba, es necesario definir las precondiciones y las postcondiciones, identificar unos valores de entrada y conocer el comportamiento que debería tener el sistema ante dichos valores.
- Tras realizar ese análisis e introducir dichos datos en el sistema, se observa si el comportamiento es el previsto o no y por qué. De esta forma se determina si el sistema ha pasado o no la prueba.
- Para cada tarea, puede surgir diversos casos de prueba, teniendo en cuenta todos los factores posibles para no dejar ningún cabo suelto sin probar y evitar así que ocurran errores no conocidos.

- Se empieza con un enfoque genérico y posteriormente se añaden más condiciones y variables que lo completen.
- Ejemplo: Tenemos una aplicación que genera un archivo de texto partiendo de los datos de un formulario.
  - 1. Definimos el caso de prueba que comprueba que el archivo de texto se genera correctamente.
  - 2. Posteriormente diversos casos de prueba anidados en el que se estipulen diferentes entradas y tipos de datos del formulario.
  - 3. Comprobar cosas como qué pasa si el archivo existen, si no existe, si existe y está vacío o tiene contenidos,…
- Podemos ver que cualquier operación sencilla, genera muchas pruebas.

- No existe una declaración oficial o formal, sobre los diversos tipos de pruebas de software. Podemos encontrarnos como dos enfoques fundamentales:
  - Pruebas de caja Blanca.
  - Pruebas de caja Negra.
- Las pruebas de caja blanca van sumamente ligadas al procedimiento en sí como una correcta evaluación de un condicional por ejemplo y las pruebas de caja negra se realizan desde el punto de vista del usuario final.
- Al final es comprobar la misma operación y los mismos datos aunque sea desde un punto de vista diferente.

2.1 CAJA BLANCA

- También son conocidas como pruebas estructurales o de caja de cristal.
- Se centran en el funcionamiento interno del programa, observando y comprobado cómo se realiza una operación.
- Son siempre las primeras pruebas que hay que realizar pues revisan la estructura y funcionalidad interna del programa. Se pretende con ello encontrar defectos básicos de software no relacionados con la interfaz del usuario. Entre otros queremos obtener casos de prueba que:
  - Garanticen que se ejecutan una vez al menos todos los caminos independientes.
  - Ejecuten todas las sentencias al menos una vez.
  - Ejecuten todos los bucles en sus límites.
  - Utilicen todas las estructuras de datos internas para asegurar su validez.

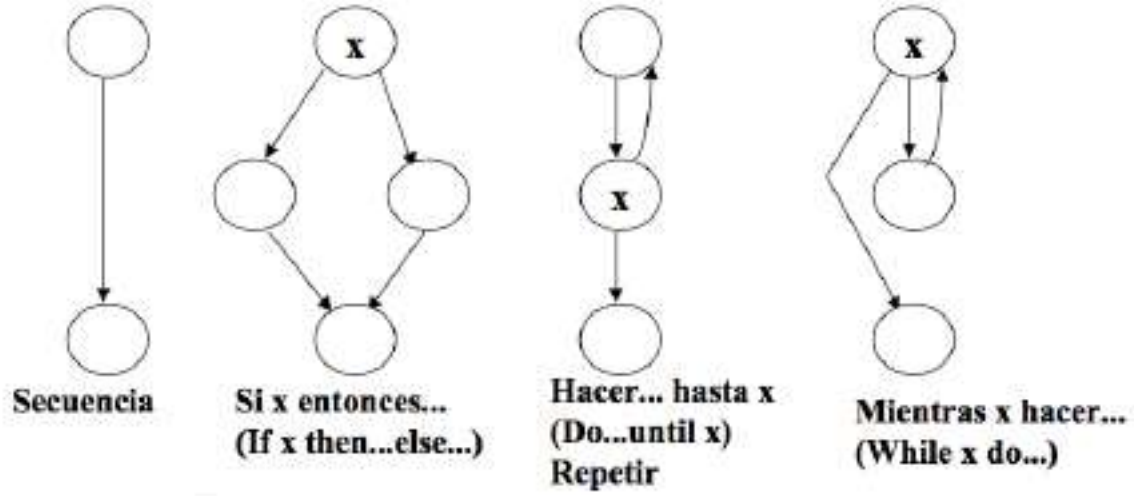
- Para ver cómo el programa se va ejecutando, y así comprobar su corrección, se utilizan este tipo de pruebas estructurales que se fijan en los caminos que se pueden recorrer.
- Se verifica la estructura de interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo.
- Con estas pruebas no pretendemos comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas las instrucciones del, que no hay código no usado, comprobar que los caminos lógicos se van a recorrer, etc.

- El diseño de caos de prueba se basa en la elección de caminos importantes que ofrezcan una seguridad aceptable de que se descubren defectos (un programa de 50 líneas con 25 sentencias if en serie da lugar a 33,5 millones de secuencias posibles), por lo que se usan los criterios de cobertura lógica siguientes ordenados de menos de menos riguroso (más barato) a más riguroso (más caro).
  - Cobertura de sentencias. Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.
  - Cobertura de decisiones/condiciones. Consisten en escribir casos suficientes para que cada decisión o condición tenga, por lo menos una vez un resultado verdadero y, al menos una vez, uno falso (incluye la obertura de sentencias).

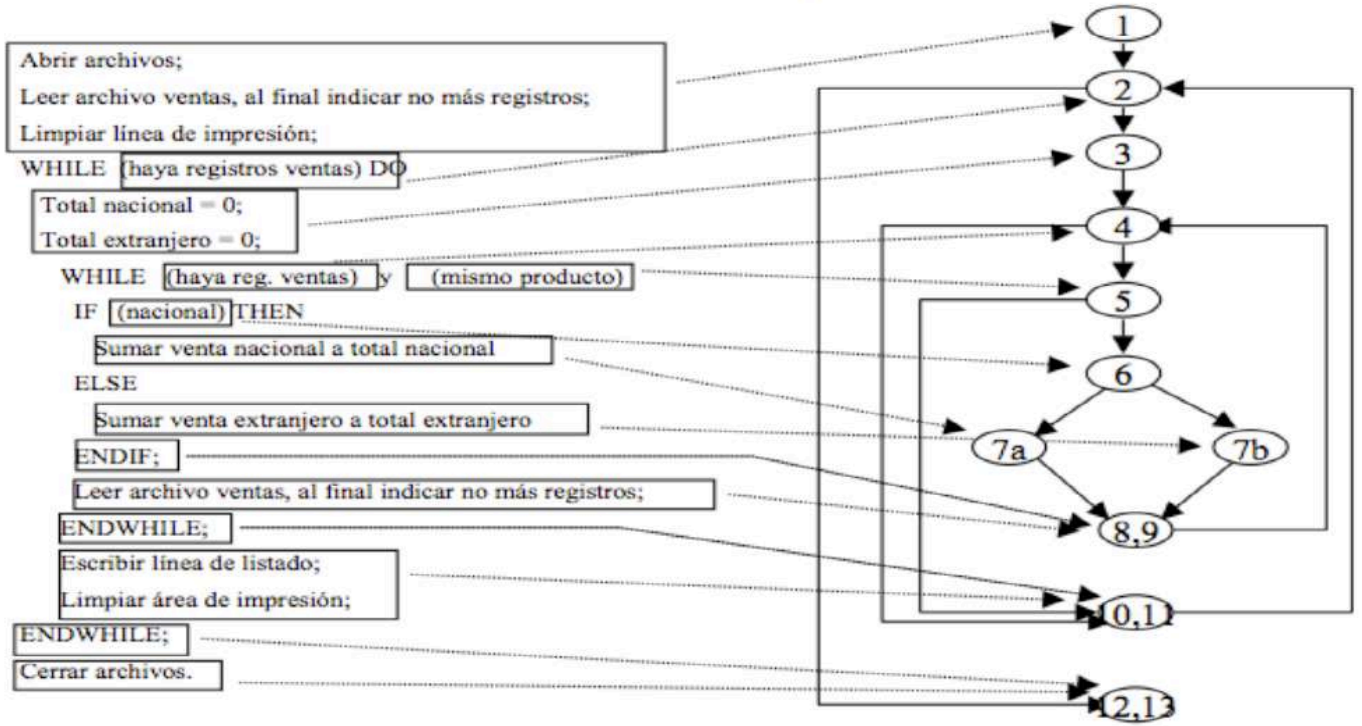
- Criterio de condición múltiple. En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea, se puede considerar que cada decisión multicondicional se descompone en varias condiciones de una condición.
- Criterio de cobertura de caminos. Se deben recorrer todos los caminos (impracticable).

- Existen diversos tipos de pruebas dentro de las pruebas de caja blanca, pudiendo medir la complejidad lógica como es la prueba del camino básico en el que se representa con un grafo de flujos con los diferentes bloques para cada camino y se obtienen los diferentes caminos para recorrerlo y así evitar saltarnos ninguna.

- Para realizar el grafo asociado a un programa (método, procedimiento, etc.) hay que realizar los siguientes pasos:
  - Separar todas las condiciones
  - Agrupar sentencias “simples” en bloques.
  - Numerar todos los bloques de sentencias y también las condiciones
  - Usar las siguientes representaciones para las diferentes sentencias de control



Ejemplo de grafo de un programa



2.2 CAJA NEGRA

- Las pruebas funcionales o de caja negra se centran en las funciones, entradas y salidas. Es impracticable probar el software para todas las posibilidades, por lo que hay que tener criterios para elegir buenos casos de prueba.
- Estas pruebas se llevan a cabo sobre la interfaz del software, no hace falta conocer la estructura interna del programa ni su funcionamiento.
- A este tipo de pruebas también se les llama prueba de comportamiento. El sistema se considera como una caja negra cuyo comportamiento solo se puede determinar estudiando las entradas y las salidas que devuelve en función de las entradas suministradas.

- Con este tipo de pruebas se intenta encontrar errores de las siguientes categorías:
  - Funcionalidades incorrectas o ausentes
  - Errores de interfaz
  - Errores en estructuras de datos o en accesos a bases de datos externas
  - Errores de rendimiento
  - Errores de inicialización y finalización.
- Existen diversos tipos de pruebas: Particiones equivalentes, análisis de valores límites o pruebas aleatorias (existen otras, pero comparte el mismo objetivo ya descrito de comprobar que los resultados son correctos).

2.2.1 PARTICIONES EQUIVALENTES

- Es un método de prueba consistente en dividir y separar los campos de entrada según el tipo de datos y las restricciones que conllevan. También se conoce como de clases equivalentes (clases de equivalencia)
- Para evitar que tengamos un exceso de pruebas, se definen unas pruebas comunes dependiendo del tipo de dato del campo en cuestión. Agrupamos los campos en diferentes baterías de pruebas, asegurándonos de realizar las pruebas de manera más completa y eficaz.
- Para identificar estas particiones o clases de equivales se examina cada condición de entrada y se divide en dos o más grupos. Generalmente se divide en dos clases de equivalencia.
  - Clases válidas: Son los valores de entrada válidos
  - Clases inválidas: Son los valores de entrada inválidos.

- La siguiente tabla resume el número de clases de equivalencia válidas y no válidas que hay que definir

Condiciones de entrada	Nª de Clases válidas	Nª de Clases inválidas
Rango	1 (valores rango)	2 (por encima y debajo del rango)
Valor específico	1 (dicho valor)	2 (por encima y debajo de ese valor)
Valor de un conjunto	1 (valores de ese conjunto)	1 (valores que no pertenecen al conjunto)
Lógica	1 (cumple la condición)	1 (no cumpla la condición)

- EJERCICIO PRÁCTICO PASO POR PASO.
- Se va a realizar una entrada de datos de un empleado por pantalla gráfica. Se definen 3 campos de entrada (el último será una lista para elegir el oficio). La aplicación acepta los datos de esta manera.
  - Empleado. Número de 3 dígitos que no empiece por 0.
  - Departamento. En blanco o número de 2 dígitos
  - Oficio. 4 valores posibles: Analista, Diseñador, Programador o Elige oficio.
- Si la entrada es correcta el programa asigna un salario (que se muestra en pantalla) según estas normas.
  - S1 si el Oficio es Analista se asigna 2500.
  - S2 si el Oficio es Diseñador se asigna 1500.
  - S3 si el Oficio es Programador se asigna 200.

- Si la entrada no es correcta el programa muestra un mensaje indicando la entrada incorrecta.
  - ER1 si el Empleado no es correcto
  - ER2 si el Departamento no es correcto
  - ER3 si no se ha elegido Oficio.
- Para representar las clases de equivalencia para cada condición de entrada se puede usar una tabla. En cada fila se definen las clases de equivalencia y añadimos un código para cada clase definida (válida y no válida) para usarlo en la definición de los casos de prueba.

Condición entrada	Clases equivalencia	Clases válidas	COD	Clases no válidas	COD
Empleado	Rango	100>=Emp<=999	V1	Emp < 100 Emp > 999	NV1 NV2
Departamento	Lógica (puede estar o no estar)	En blanco	V2	No es un número	NV3
	Valor	Cualquier número de dos dígitos	V3	Número de más de 2 dígitos Número de menos de 2 dígitos	NV4 NV5
Oficio	Miembro de un conjunto	Oficio = programador	V4	Oficio = Elige otro oficio	NV6
		Oficio = Analista	V5		
		Oficio = Diseñador	V6		

- A partir de esta tabla se generan los casos de prueba. Utilizaremos las condiciones de entrada y las clases de equivalencia.
- Para ello se representará otra tabla donde cada fila representa un caso de prueba con los códigos de las clases de equivalencia que se aplican, los valores asignados a las condiciones de entrada y el resultado esperado según el enunciado del problema.
- Tendremos casos de prueba con resultados válidos (CP1, CP2, CP3, CP4) y otros que nos tienen que dar resultados no válidos (CP5, CP6, CP7 o CP8).
- Se irían añadiendo a la tabla todas las combinaciones posibles de clases de equivalencia válidas y no válidas (en la tabla por cuestiones de espacio faltan algunas como (V1, V2, V6) y (V1, V3, V5) que darían una equivalencia válida y otras no válidas como (NV2, V2, V4).

Caso prueba	Clases equivalencia	Condiciones de Entrada			Resultado esperado
		Empleado	Departamento	Oficio	
CP1	V1, V3, V4	200	20	Programador	S3
CP2	V1, V2, V5	250		Analista	S1
CP3	V1, V3, V6	450	30	Diseñador	S2
CP4	V1, V2, V4	220		Programador	S3
CP5	NV1, V3, V6	90	35	Diseñador	ER1
CP6	V1, NV3, V5	100	AD	Analista	ER2
CP7	V1, V2, NV6	300		Elige oficio	ER3
CP8	V1, NV4, V6	345	123	Diseñador	ER2
.....					

- EJERCICIO
- Tenemos una función que se llama parVocal que va a devolver un String que indique “Par y Vocal”, “Par y No Vocal” o “Error no es Par” como resultado a las diferentes combinaciones de entrada de los dos parámetros que se metan.
  - Si se mete un par en el primer parámetro y una vocal -> “Par y Vocal”
  - Si se mete un par pero no es una vocal -> “Par y no Vocal”
  - Si no mete un par en el primer parámetro directamente -> “Error no es par” con independencia de que el segundo parámetro sea o no vocal
- Según el método de particiones o clases de equivalencia, ¿qué tipo de pruebas deberíamos realizar?

- SOLUCIÓN
  - Solo tenemos dos condiciones de entrada (número y una cadena). Tendré por tanto dos clases de equivalencia de tipo valor para el primero y miembro de un conjunto para el segundo (vocales)
  - Valores correctos
    - S1. Si mete un par y vocal que devuelva la cadena “Par y Vocal”
  - Resultado erróneos
    - ER1. Meter un número par y un no vocal “Par y no vocal”
    - ER2. Meter un número no par “Error no es un número”

Condición	Clase	Válida	COD	No Válida	COD
Número	Valor	Cualquier número par	V1	Número impar Cadena	NV1 NV2
Letra	Miembro	Cualquier vocal	V2	Cualquier no vocal	NV3

Caso prueba	Clases de equivalencia	CONDICIONES DE ENTRADA		Resultado
		Número	Vocal	
CP1	V1, V2	202	I	S1
CP2	V1, NV3	604	R	ER1
CP3	NV1, V2	333	A	ER2
CP4	NV1, NV3	215	S	ER2
CP5	NV2, V2	M	E	ER2
CP6	NV2, NV3	P	M	ER2

2.2.2 ANÁLISIS DE VALORES LÍMITE

- Es una técnica complementaria a la partición equivalente, básicamente, nos indica que si especificamos un rango de valores o un número de valores específicos, también se deberá probar por el valor inmediatamente superior e inferior de dichos cotas.
- Esto se basa en que los errores tienden a producirse con más probabilidad en los límites o extremos de los campos de entrada.
- Reglas
  - 1. Si es de tipo rango de valores, se deben diseñar casos de prueba para los límites del rango y para los valores justo por encima y por debajo del rango. Por ejemplo, si una entrada requiere un rango de valor entre 1 y 10, haremos un caso de prueba para el 1, 10, 0 y 11.

- 2. Si una condición de entrada especifica un "número de valores", se deben diseñar casos de prueba que ejerciten los valores máximo, mínimo, un valor justo por encima del máximo y un valor justo por encima del mínimo. Por ejemplo si requiere un valor de dos a diez datos de entrada, hay que escribir casos de prueba para 2, 10, 1 y 11 datos de entrada.
- 3. La regla 1 se puede usar para la condición de salida. Por ejemplo, si se debe aplicar sobre un campo de salir un descuento de entre un 10% y un 50% máximo, tratar de generar casos de prueba que generarán un valor del 9.99%, 10%, 50%, 50.01%.
- 4. La regla se puede usar para la condición de salida, si se va a sacar una tabla de 10 elementos, diseñar casos de prueba que produzcan 0, 1, 10 y 11 elementos.
- 5. Si las estructuras de datos internas tienen límites prestablecidos (por ejemplo un array de 100 elementos), hay que asegurarse de diseñar casos de prueba que ejercite esa estructura de datos.

- Ejemplo. Si partimos de nuestro caso práctico del empleado que era un número de tres dígitos de 100 a 999. Utilizando esta técnica podemos diseñar nuevos casos prácticos con el límite inferior y superior del rango (podemos indicar V1a el valor100, V1b el valor 999 y NV1a el valor 99 y NV1b el valor 1000).

Caso prueba	Clase equivalencia	Condiciones de entrada			Resultado esperado
		Empleado	Departamento	Oficio	
CP11	V1a, V3, V4	100	20	"Programador"	S3
CP12	V1b, V2, V5	999		"Analista"	S1
CP13	NV1a, V3, V6	99	30	"Diseñador"	ER1
CP14	NV1b, V2, V4	1000		"Programador"	ER1

2.3 OTROS TIPOS DE PRUEBAS

- Se pueden realizar otros tipos de pruebas que midan por ejemplo el rendimiento o la coherencia.
- En las pruebas de rendimiento se mide el tiempo que le ha tomado a la aplicación realizar una acción específica. Si bien esto depende de otros factores (como la máquina) sigue teniendo cierta validez, puesto que nos permite comprobar y controlar ese tiempo en ese equipo concreto y tratar de conseguir un mejor rendimiento.
- Para estas mediciones, normalmente los lenguajes de programación pueden tener algunos herramientas de medida como sencillos cronómetros (en .NET o JAVA por ejemplo existe una clase StopWatch)

```
static void Main(string[] args) {
            IList<int> lista = new List<int>();
            IList<int> lista2 = new List<int>();
            for (int i = 0; i < 1000; i++)
                lista.Add(i);
            Stopwatch crono = new Stopwatch();
            crono.Start();
            foreach (int num in lista)
                if (num % 2 == 0)
                    lista2.Add(num);
            crono.Stop();
            TimeSpan duracion1 = crono.Elapsed;

        crono.Reset();
            crono.Start();
            lista2 = lista.Where(n => n % 2 == 0).ToList();
            crono.Stop();
            TimeSpan duracion2 = crono.Elapsed;


            System.Console.WriteLine("Tiempo 1 : {0}\n",duracion1);
            System.Console.WriteLine("Tiempo 2 : {0}",duracion2);
         }
```

- Las pruebas de coherencias son subjetivas, es decir, no están ligadas a la propagación ni a los datos en sí mismos.
- Con ellas comprobamos si la funcionalidad de la aplicación es correcta y no si la aplicación funciona correctamente.
- Por ejemplo, si tenemos empleados y pedidos de restaurante de comida a domicilio. Los empleados pueden ser de tipo cocinero o repartidor. Si para repartir un pedido nos asigna a un empleado que es cocinero, el resultado de la aplicación no es coherente.
- Está muy ligado a las pruebas de caja negra, ya que deberíamos filtrar correctamente el tipo de empleado para elegir en la aplicación.

- Los últimos casos de pruebas que se pueden hacer son las aleatorias. Y la conjetura de errores.
- En las pruebas aleatorias, simulamos la entrada habitual del programa creando datos d entrada con algún tipo de generador automático de casos de prueba.
- En la conjetura de errores se enumera una lista de posibles equivocaciones típicas de los desarrolladores y situaciones propensas a error.
  - Valor 0 es una situación propensa de error
  - No meter ningún valor en la variable en parámetros de un solo valor
  - Etc.

3. HERRAMIENTAS DE DEPURACIÓN

- El proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados de la ejecución y fruto de esa evaluación se comprueba que hay (o no) una falta de correspondencia entre los resultados esperados y los obtenidos realmente.
- Al desarrollar cometemos dos tipos de errores: errores de compilación y errores lógicos.
  - Los primeros son fáciles de corregir ya que normalmente un IDE nos suele proporcionar información de la localización del error e incluso cómo solucionarlo.
  - Los errores de tipo lógico son más difíciles de detectar ya que el programa compila con éxito, sin embargo, su ejecución puede devolver resultados inesperados y erróneo. A estos errores de tipo lógico se suelen llamar bugs.

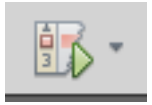
- Los IDEs modernos incluyen herramientas de depuración que permiten verificar el correcto funcionamiento del código generado.
- El depurador nos permite analizar el código del programa mientras se ejecuta, por eso se suele decir que depurar un programa es el proceso de encontrar y resolver los defectos para conseguir el funcionamiento correcto del programa mediante la ejecución controlada del software.
- En ocasiones, los IDEs, también son capaces de detectar otros pequeños problemas que no son errores pero requieren la atención del programador
  - Warnings → Advertencias
  - Permiten que el programa se compile y ejecute.

- Para ello vamos a poder establecer puntos de interrupción o ruptura, suspender la ejecución del programa, ejecutar el código paso a paso y/o examinar el contenido de las variables (entre otras muchísimas opciones disponibles).
- Por tanto podemos decir que los depurados nos ayudan a realizar pruebas
  - Funciones. Hace lo que tiene que hacer
  - Estructurales. El código es bueno, no hay código inalcanzable, se utilizan adecuadamente los recursos, etc.

3.1 EL DEBUGGER DE NETBEANS

- El debugger o depurador de NetBeans permite:
  - Ejecutar el código fuente paso a paso
  - Ejecutar métodos del JDK paso a paso.
  - Utilizar breakpoint para detener la ejecución del programa y poder observar el estado de las variables.
  - Conocer el valor que toma cada variable o expresión según se van ejecutando las líneas de código.
  - Modificar el valor de una variable sobre la marcha y continuar la ejecución.
- Mediante el debugger podemos:
  - Encontrar de forma rápida y fácil errores en el programa.
  - Entender mejor el flujo de ejecución del programa.

- Hay varias formas de iniciar la ejecución del programa en modo depuración:
  - **1: Depurar (Debug) → Debug Project** (Ctrl + F5 en Windows) o pulsando sobre el botón:

    
  - También pulsando con el botón derecho sobre un proyecto en explorador de proyectos y seleccionar Depurar (Debug)
  - El programa se ejecuta hasta llegar al primer breakpoint. Si no han establecido breakpoints, el programa se ejecutará normalmente hasta el final.

- **2: Depurar (debug) → Ejecutar hasta el cursor (Run to Cursor)** (F4)
  - Se ejecuta el programa hasta la instrucción donde se encuentra el cursor.

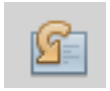
- **3: Depurar (Debug) → Paso a paso (Step Into)** (F7)
  - Comienza la depuración desde la primera línea del método main. El depurador se detiene esperando que decidamos el modo de depuración.

- Una vez iniciada la depuración y el depurador se detiene, la siguiente línea de código que se va a ejecutar aparece en verde, con una flecha verde a su izquierda

```
/**
 *
 * @author guillermopalazoncano
 */
public class PruebaDepuracion {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int a = 1;
        System.out.println(a);
        a++;
        System.out.println(a);
        a++;
```

- En este punto la depuración puede continuar utilizando distintas opciones:



- 1. Continuar ejecución (Step Over) - F8. Ejecuta una línea de código. Si la instrucción es una llamada a un método, ejecuta el método sin entrar dentro del código del método.

```
/**
 *
 * @author guillermopalazoncano
 */
public class PruebaDepuracion {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int a = 1;
        System.out.println(a);
        a++;
        System.out.println(a);
```

- 2. Paso a Paso (Step Into) - F7. Ejecuta una línea de código. Si la instrucción es una llamada a un método, salta al método y continúa la ejecución por la primera línea del método

- 3. Ejecutar y salir (Step Out) - Ctrl + F7. Ejecuta una línea de código. Si la línea de código actual se encuentra dentro de un método, se ejecutarán todas las instrucciones que queden del método y se vuelve a la instrucción desde la que se llamó al método.

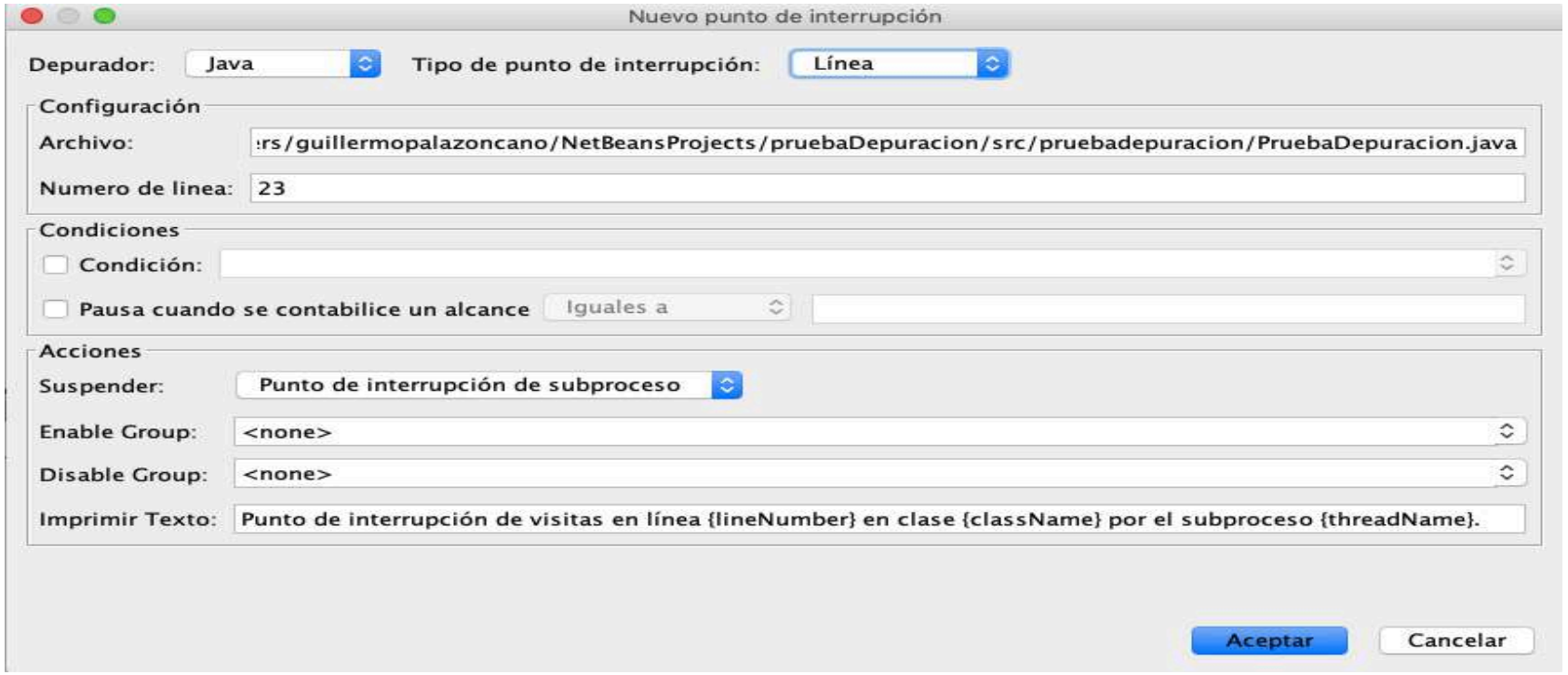
- 4. Ejecutar hasta el cursor (Run to Cursor) - F4. Se ejecuta el programa hasta la instrucción donde se encuentra el cursor.

- 5. Continuar (Continue) - F5. La ejecución del programa continúa hasta el siguiente breakpoint. Si no existe un breakpoint se ejecuta hasta el final.

- 6. Finalizar sesión del depurador (Finish Debugger Session) - Mayúsculas + F5. Termina la depuración del programa.

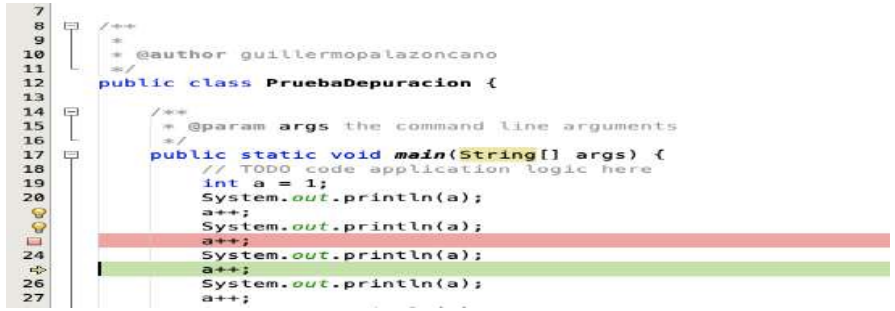
3.2 PUNTOS DE RUPTURA

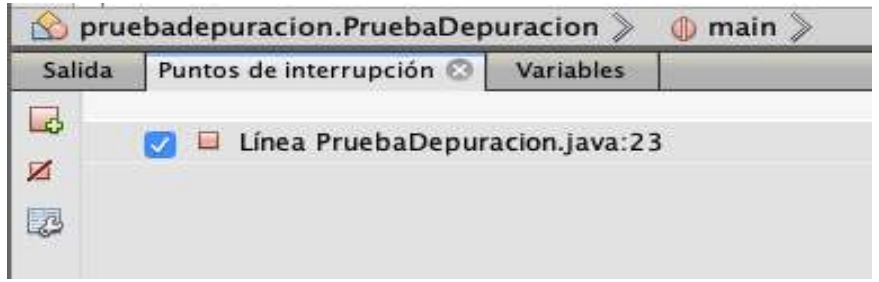
- Dentro del menú de depuración nos encontramos con la opción de "Nuevo punto de interrupción" (inserta punto de ruptura o breakpoint).
- Existen diferentes tipos de puntos de interrupción, generalmente utilizaremos el de línea (aunque debemos llevar cuidado porque por defector viene elegido el de método)
- Para ello se selecciona la línea de código donde queremos que el programa se pare, para a partir de ella, inspecciones 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, también conocidos como breakpoints


Nuevo punto de interrupción
Depurador:
Java
Tipo de punto de interrupción:
Línea
Configuración
Archivo:
:rs/guillermopalazoncano/NetBeansProjects/pruebaDepuracion/src/pruebadepuracion/PruebaDepuracion.java
Numero de linea:
23
Condiciones
Condición:
Pausa cuando se contabilice un alcance
Iguales a
Acciones
Suspender:
Punto de interrupción de subproceso
Enable Group:
<none>
Disable Group:
<none>
Imprimir Texto:
Punto de interrupción de visitas en línea {lineNumber} en clase {className} por el subproceso {threadName}.
Aceptar
Cancelar

- Pueden establecerse en cualquier línea de código ejecutable (no sería válido un comentario, o una línea en blanco).
- Una vez insertado el puntura, e iniciada la depuración, el programa a evaluar se ejecutaría hasta la línea marcada por 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 puede iniciar una depuración paso a paso, e ir comprobando el camino que toma el programa a partir del punto de ruptura.
- Una vez realizada 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.

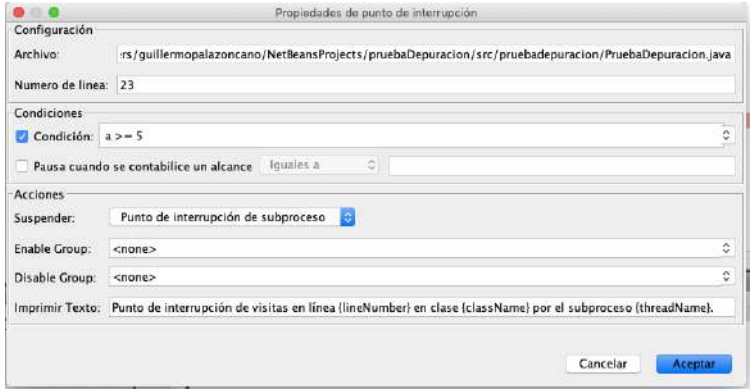
- Cuando se fija un breakpoint en Netbeans queda resaltado en rojo.

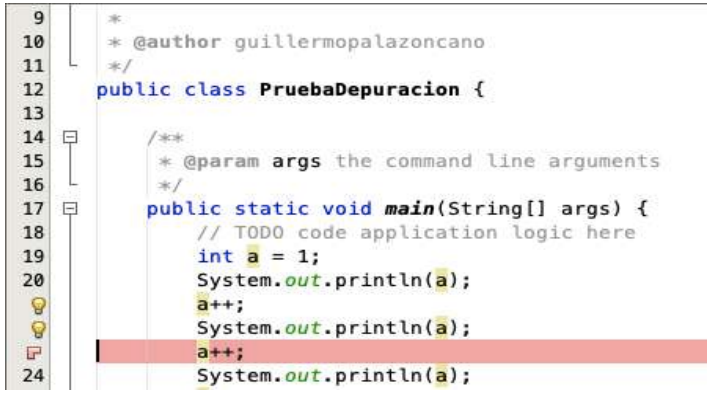




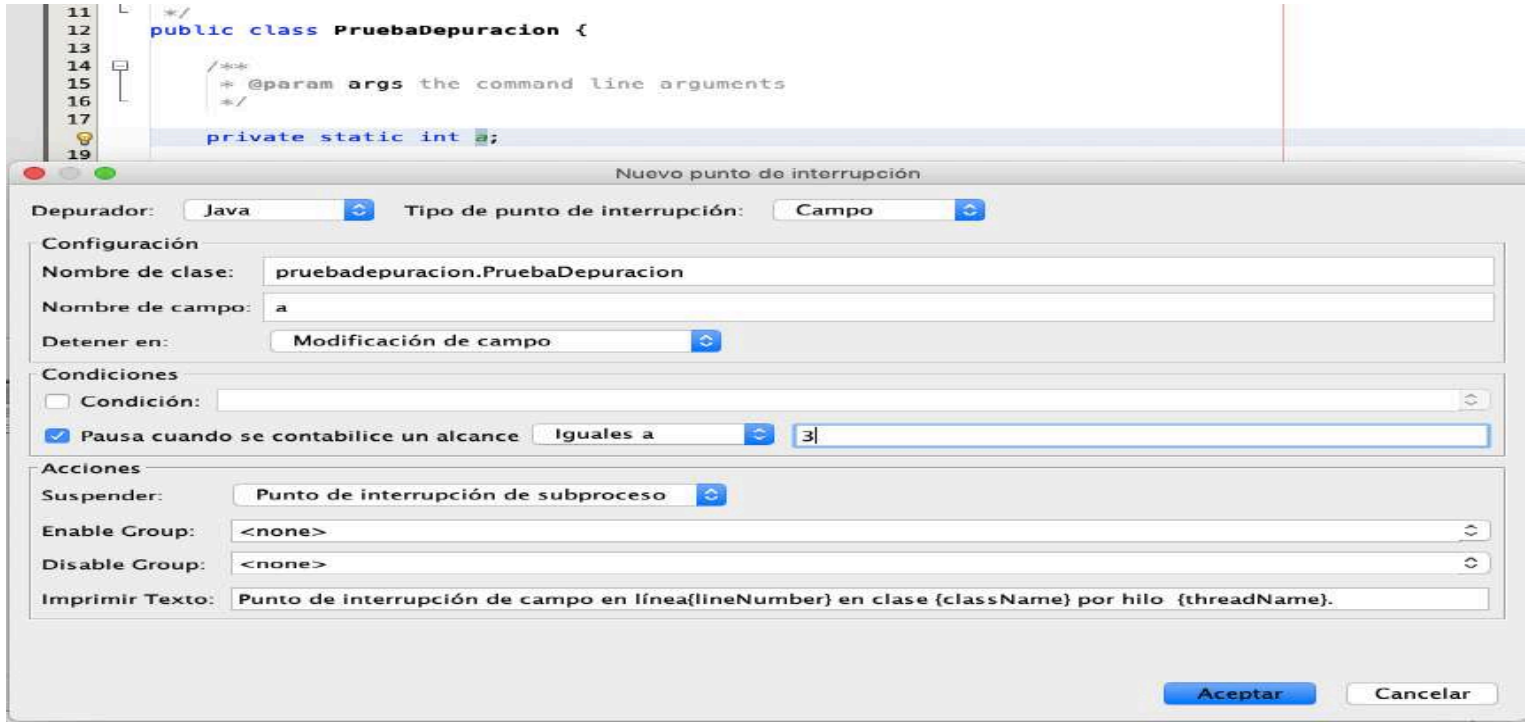
- Para eliminar un breakpoint se pulsa sobre el cuadrado rojo.
- Para eliminarlos todos: botón derecho en la ventana de breakpoints -> Delete All
- Para desactivar un breakpoint, botón derecho sobre la marca roja -> breakpoint -> desmarcar Enable.
- Para desactivarlos todos: botón derecho en la ventana de breakpoints -> Disable All
- Los breakpoints desactivados aparecen en gris.

- Podemos hacer que la depuración se detenga en un breakpoint solo si se cumple una determinada condición o si la línea de código se ha ejecutado un número determinado de veces. Para fijar un breakpoint condicional:
  - Botón derecho sobre el breakpoint -> Breakpoint > Properties
  - La marca cambia de aspecto (al cuadrado rojo le falta la esquina inferior derecha)

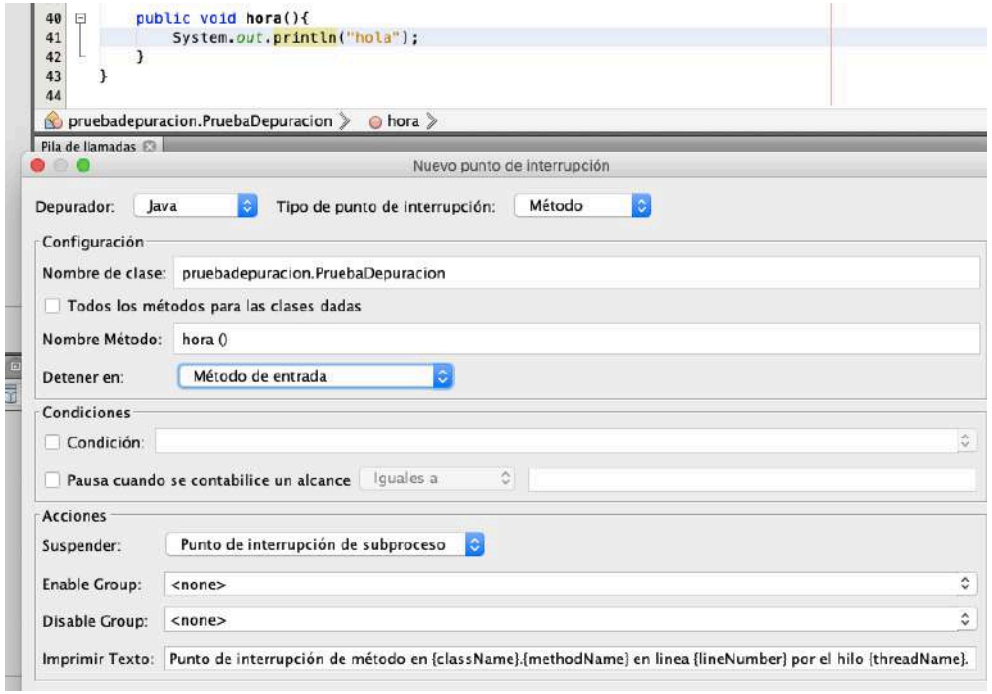




- En NetBeans podremos poner un punto de interrupción también para cuando ocurra algo sobre un campo determinado (en Eclipse y otros IDEs esto se conoce como WatchPoint). Seleccionaremos la variable e indicaremos un punto de interrupción sobre ella de tipo Campo. Ojo, esta variable tiene que estar definida a nivel de clase.
  - Cuando la variable sea leída → Accesso.
  - Cuando sea escrita → Modificación
  - Ambas cosas.

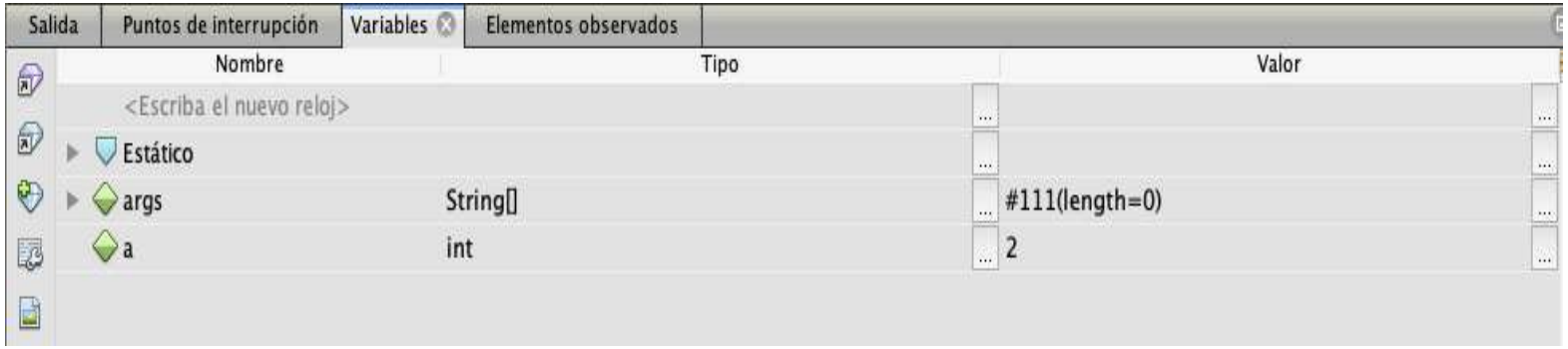

11 */
12 public class PruebaDepuracion {
13
14 /**
15 * @param args the command line arguments
16 */
17
19 private static int a;
Nuevo punto de interrupción
Depurador: Java
Tipo de punto de interrupción: Campo
Configuración
Nombre de clase: pruebadepuracion.PruebaDepuracion
Nombre de campo: a
Detener en: Modificación de campo
Condiciones
Condición:
Pausa cuando se contabilice un alcance
Iguales a
3
Acciones
Suspender: Punto de interrupción de subproceso
Enable Group: <none>
Disable Group: <none>
Imprimir Texto: Punto de interrupción de campo en línea{lineNumber} en clase {className} por hilo {threadName}.
Aceptar
Cancelar

- También se puede realizar un punto de interrupción cuando se accede a algún método.
  - Cuando se entre al método
  - Cuando se salga del método
  - Cuando se entre o salga un número de veces
  - Cuando ocurra alguna condición.



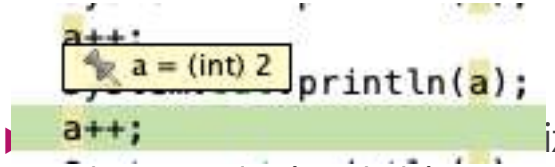
- En el proceso de depuración se usan distintas ventanas situadas bajo la ventana de código. Algunas aparecen automáticamente.
- Para mostrarlas pulsar Ventanas > Depuración y seleccionar la que queramos.
- Las más importantes son: Puntos de interrupción, Variables y Elementos Observados (Watches).
- La pila de llamadas permite ver la pila actual de llamadas a métodos, es decir, qué método llamo al actual y sucesivamente

- Ventana de variables locales
  - En esta ventana se muestran las variables, su tipo y su valor actual.



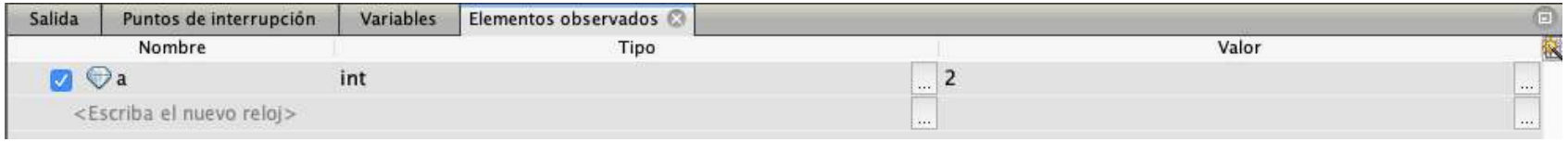
  - El debugger permite cambiar el valor de una variable local en esta ventana y continuar la ejecución del programa usando el nuevo valor de la variable.

- Ventana de Elementos Observados
  - Para obtener el valor de una variable durante la depuración de un programa se sitúa el ratón sobre la variable y se mostrará, el depurador mostrará el tipo y el valor de la variable.



  - izar el valor de una variable es mediante un elemento observado (watch).
  - Para agregar un watch a una variable o una expresión, selecciona la variable o expresión a supervisar y a continuación pulsar. Depurar → Nuevo elemento observado.
  - O también: botón derecho sobre la variable → Nuevo elemento observado.

- Las variables a las que se les ha asignado un elemento observado aparecen en la ventana Elementos Observados:



- La barra de herramientas del depurador aparece cuando iniciamos la depuración. Para mostrarla siempre: Ver → Barra de Herramientas → Depurar.
- Vídeo completo depuración: https://www.youtube.com/watch?v=xCvoSnR-PQg