

Tema3. Diseño y realización de pruebas

IES PEDRO MERCEDES

CUENCA 2019/20

MARINA MORENO MARTÍNEZ

1. Introducción

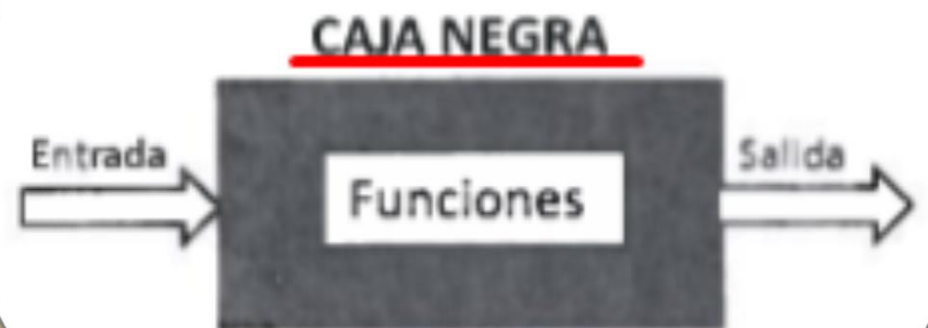
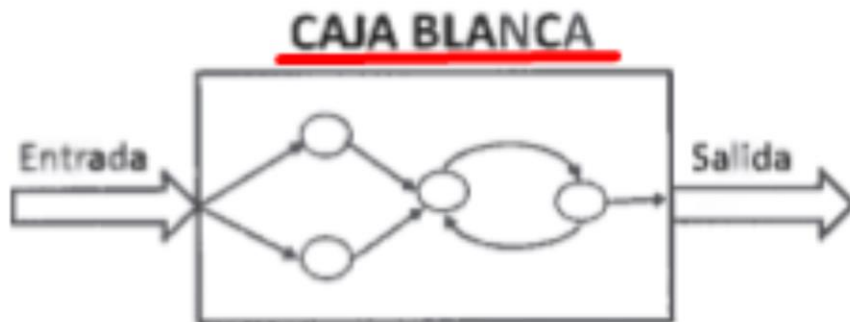
- Las pruebas de software consisten en verificar y validar un producto software antes de su puesta en marcha.
- Constituyen una de las etapas del desarrollo de software, y básicamente consiste en probar la aplicación construida.
- Se integran dentro de las diferentes fases del ciclo de vida del software dentro de la ingeniería de software.

2. Técnicas de diseño de pruebas

- Etapas de la ejecución de pruebas:
 - ✓ Planificación de pruebas
 - ✓ Diseño y construcción de los casos de prueba
 - ✓ Definición de los procedimientos de prueba
 - ✓ Ejecución de las pruebas
 - ✓ Registro de resultados obtenidos
 - ✓ Registro de errores encontrados
 - ✓ Depuración de los errores e informe de los resultados obtenidos.

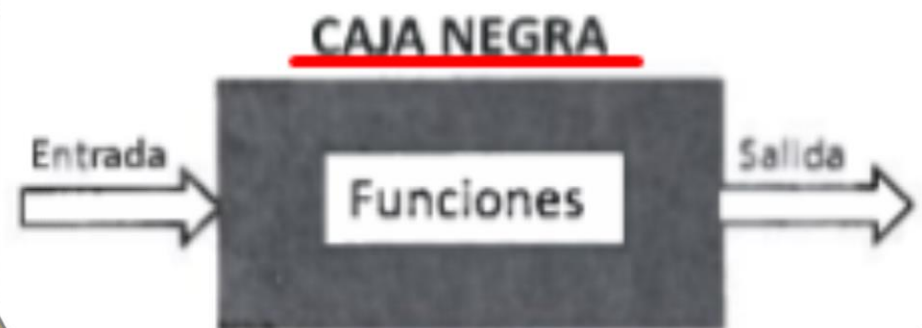
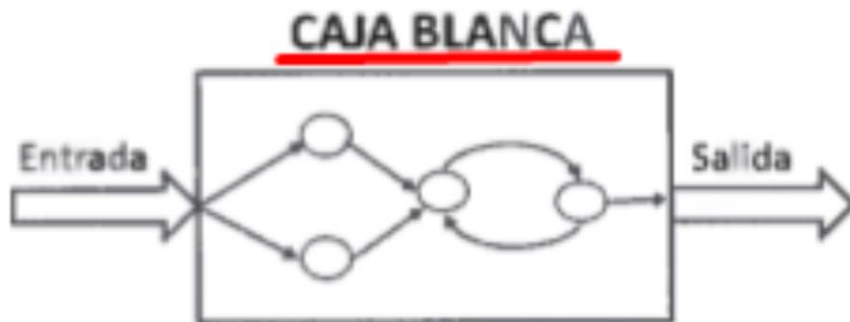
2. Técnicas de diseño de pruebas

- **Caso de prueba** conjunto de entradas, condiciones de ejecución y resultados esperados de salida para conseguir un objetivo particular o condición de prueba.
- Necesitamos **precondiciones** y **postcondiciones**.
- Técnicas o enfoques:



2. Técnicas de diseño de pruebas

- **Caso de prueba** conjunto de entradas, condiciones de ejecución y resultados esperados de salida para conseguir un objetivo particular o condición de prueba.
- Necesitamos **precondiciones** y **postcondiciones**.
- Técnicas o enfoques:



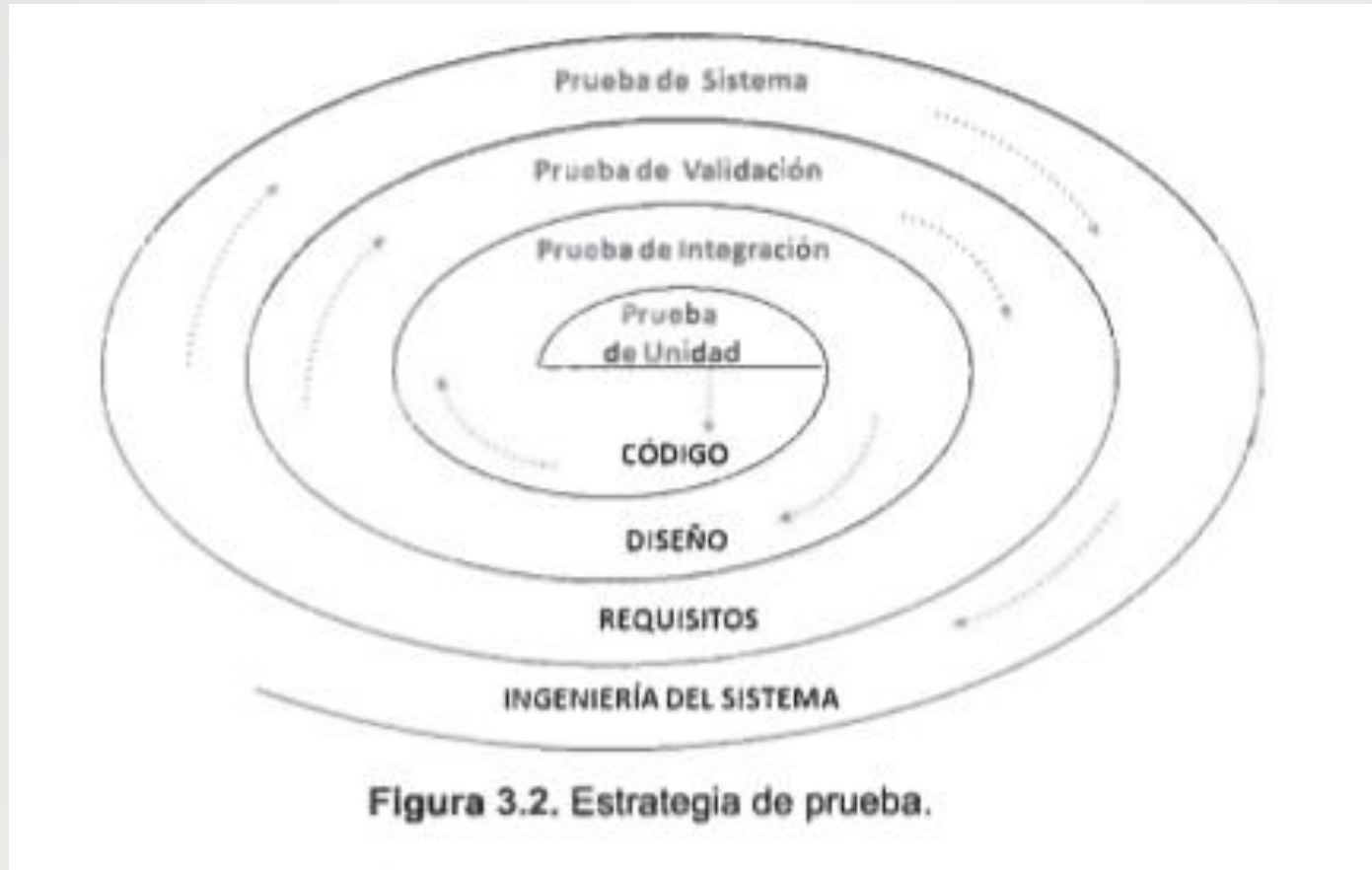
2.1 Pruebas de caja blanca

- **Pruebas estructurales o de caja de cristal**
- Se centran **en estudiar minuciosamente el código** de la aplicación.
- Casos de prueba que:
 - ✓ Garanticen que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
 - ✓ • Ejecuten todas las sentencias al menos una vez.
 - ✓ • Ejecuten todas las decisiones lógicas en su parte verdadera y en su parte falsa.
 - ✓ • Ejecuten todos los bucles en SUS límites.
 - ✓ • Utilicen todas las estructuras de datos internas para asegurar su validez.
- Ejemplo: **Prueba del camino básico.**

2.1 Pruebas de caja negra

- También se les llama **prueba de comportamiento**.
- Se llevan a cabo sobre la interfaz del software.
- Comprueban que las salidas que devuelve la aplicación son las esperadas en función de las entradas que se proporcionen.
- Se intenta encontrar errores de las siguientes categorías:
 - ✓ Funcionalidades incorrectas o ausentes.
 - ✓ Errores de interfaz.
 - ✓ Errores en estructuras de datos en accesos a bases de datos externas.
 - ✓ Errores de rendimiento.
 - ✓ Errores de inicialización y finalización.
- Ejemplo: **Clases de equivalencia, análisis de valores límite.**

3. Estrategias de pruebas del Software



3. Estrategias de pruebas del Software

3.1 Prueba de unidad

- Se prueba cada unidad o módulo con el objetivo de eliminar errores en la interfaz y en la lógica interna.
- Se utilizan técnicas de caja negra y caja blanca.
- Se prueba:
 - ✓ La interfaz del módulo.
 - ✓ Las estructuras de datos locales.
 - ✓ Las condiciones límite.
 - ✓ Todos los caminos independientes de la estructura de control.
 - ✓ Todos los caminos de manejo de errores.

3. Estrategias de pruebas del Software

3.1 Prueba de unidad



Figura 3.3. Prueba de unidad.

3. Estrategias de pruebas del Software

3.1 Prueba de integración

- Se prueba como interaccionan los distintos módulos.
- Hay dos enfoques fundamentales:
 - ✓ **Integración no incremental o big bang.**
 - ✓ **Integración incremental.** Estrategia ascendente y descendente.

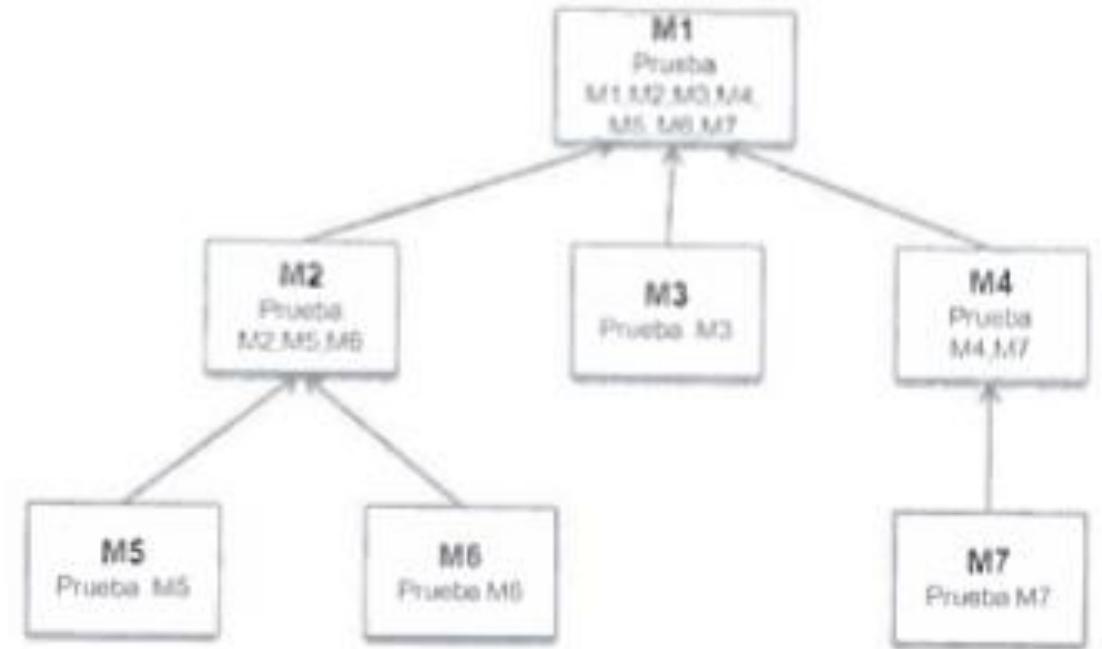


Figura 3.4. Prueba de integración Ascendente.

3. Estrategias de pruebas del Software

3.1 Prueba de validación

- La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente definidas en el documento de especificación de requisitos del software o ERS.
- Se llevan a cabo una serie de pruebas de caja negra.
- Las técnicas utilizadas son:
 - ✓ **Prueba Alfa.** El cliente prueba el software en el lugar de desarrollo.
 - ✓ **Prueba Beta.** El cliente prueba el software en el lugar de trabajo.

3. Estrategias de pruebas del Software

3.1 Prueba de sistema

- La prueba del sistema está formada por un conjunto de pruebas cuya misión es ejercitar profundamente el software.
- Son las siguientes:
 - ✓ **Prueba de recuperación.**
 - ✓ **Prueba de seguridad.** Accesos ilegales.
 - ✓ **Prueba de resistencia (Stress).** Probar el programa bajo una gran demanda de recursos.

4. Documentación para las pruebas

- El estándar IEEE829-1998 describe el conjunto de documentos que pueden producirse durante el proceso de prueba.
- Son los siguientes:
 - ✓ **Plan de Pruebas.**
 - ✓ **Especificaciones de prueba** que contiene:
 - ✓ Diseño de las pruebas
 - ✓ Casos de prueba
 - ✓ Procedimientos de prueba
 - ✓ **Informes de Prueba.**

5. Pruebas de código

- La prueba del código consiste en la ejecución del programa(o parte de él)con el objetivo de encontrar errores.
- Se parte para su ejecución de un conjunto de entradas y una serie de condiciones de ejecución se observan y registran los resultados y se comparan con los resultados esperados.

5. Pruebas de código

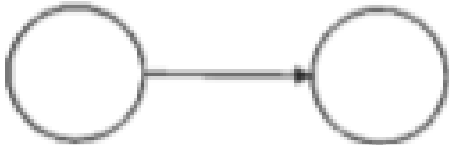
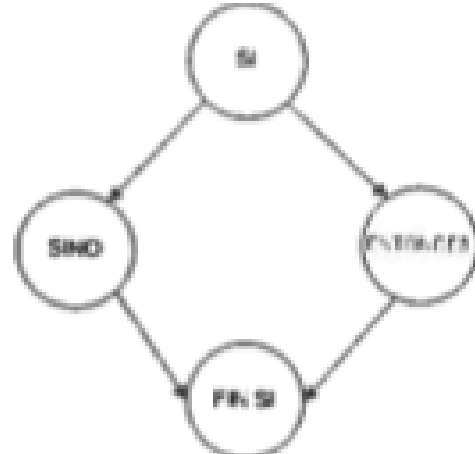
5.1 Prueba del camino básico

- Prueba de caja blanca.
- Obtienes la **complejidad ciclomática** de un proceso y a partir de ella se definen un conjunto básico de caminos de ejecución.
- Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se **ejecuta por lo menos una vez cada sentencia del programa.**
- Para la obtención de la medida de la complejidad lógica (o complejidad ciclomática) emplearemos una representación del flujo de control denominada **grafo de flujo o grafo del programa.**

5. Pruebas de código

5.1 Prueba del camino básico

- NOTACIÓN DE GRAFO DE FLUJO

ESTRUCTURA	GRAFO DE FLUJO
SECUENCIAL Instrucción 1 Instrucción 2 Instrucción n	
CONDICIONAL Si <condición> Entonces <Instrucciones> Si no <Instrucciones> Fin si	

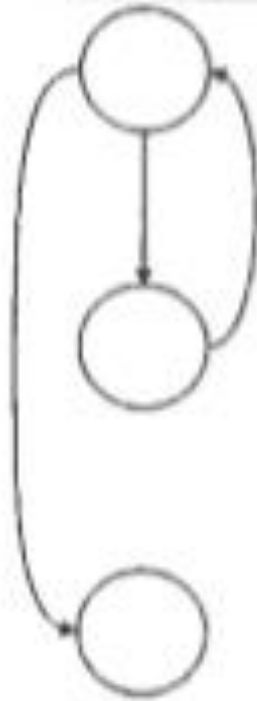
5. Pruebas de código

5.1 Prueba del camino básico

- NOTACIÓN DE GRAFO DE FLUJO

HACER MIENTRAS

Mientras <condición> Hacer
 <instrucciones>
Fin mientras



REPETIR HASTA

Repetir
 <instrucciones>
Hasta que <condición>



5. Pruebas de código

5.1 Prueba del camino básico

- NOTACIÓN DE GRAFO DE FLUJO

CONDICIONAL MÚLTIPLE

Según sea <variable> **Hacer**

Caso opción 1:

<Instrucciones>

Caso opción 2:

<Instrucciones>

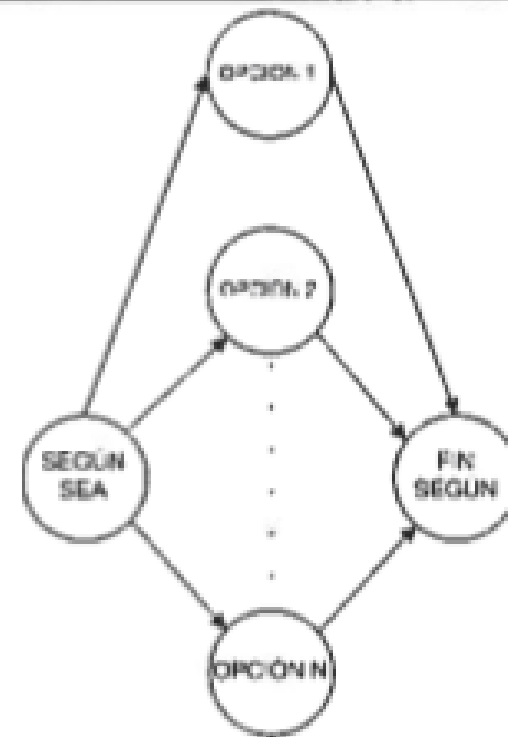
Caso opción 3:

<Instrucciones>

Otro caso:

<Instrucciones>

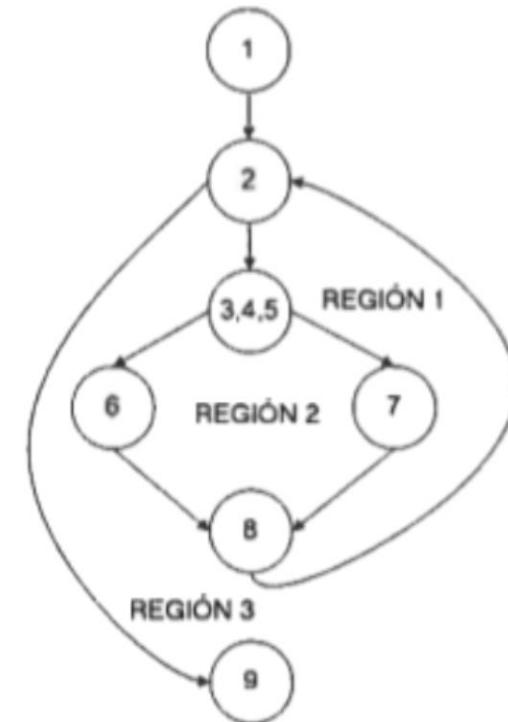
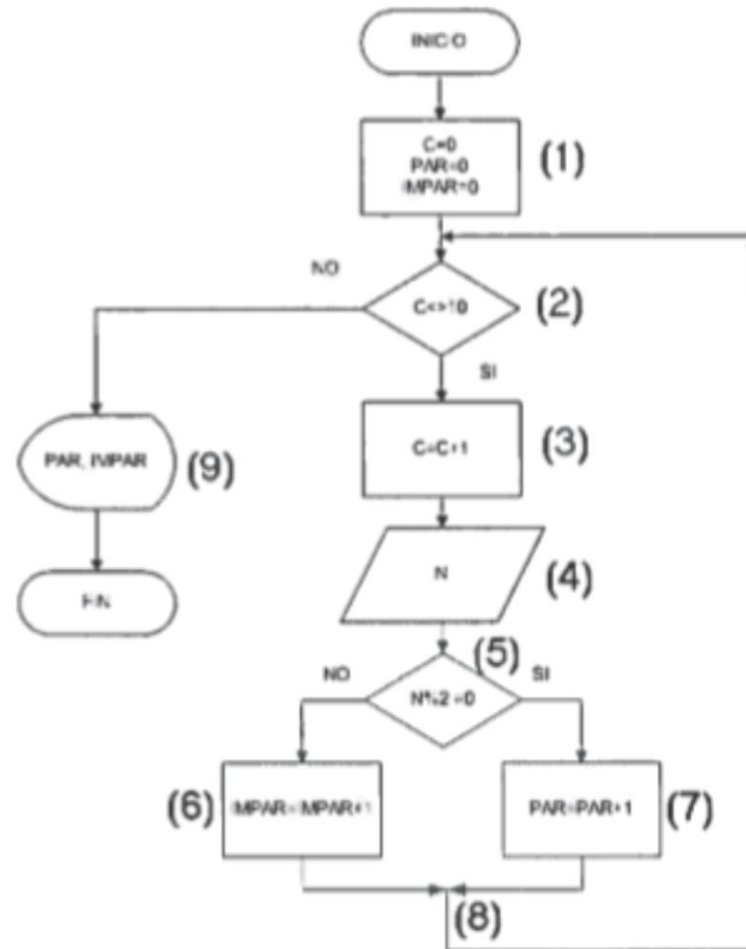
Fin según



5. Pruebas de código

5.1 Prueba del camino básico

- EJEMPLO:



5. Pruebas de código

5.1 Prueba del camino básico

- EJEMPLO 2:

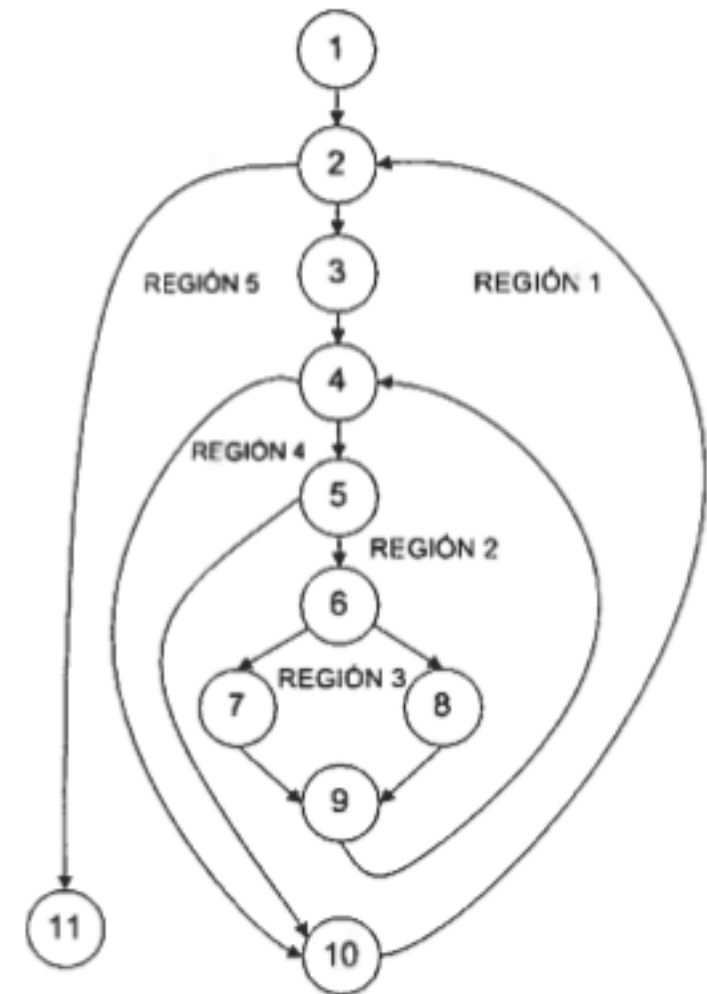
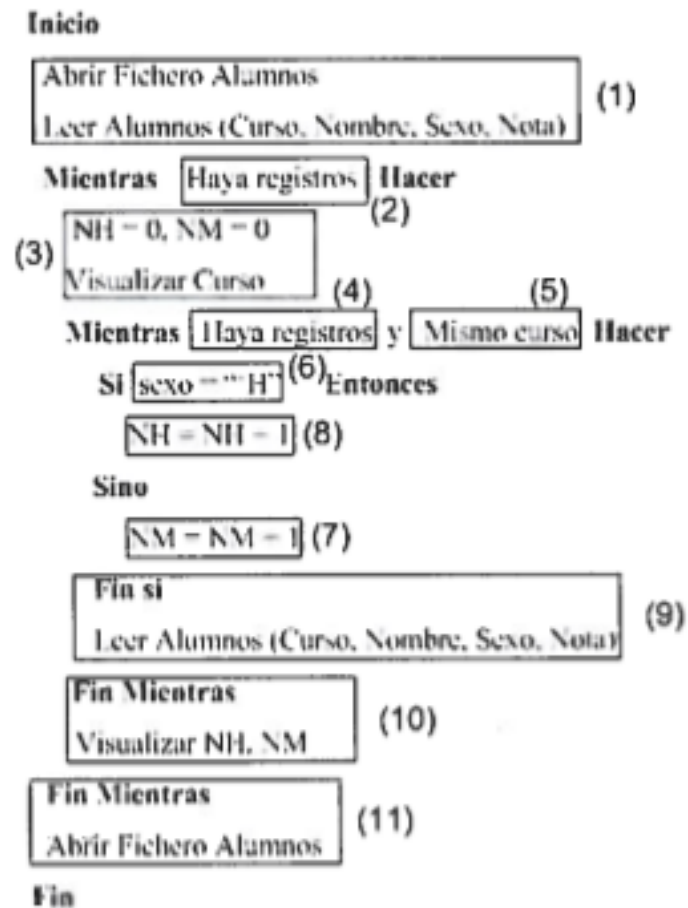


Figura 3.6. Pseudocódigo y grafo de flujo.

5. Pruebas de código

5.1 Prueba del camino básico

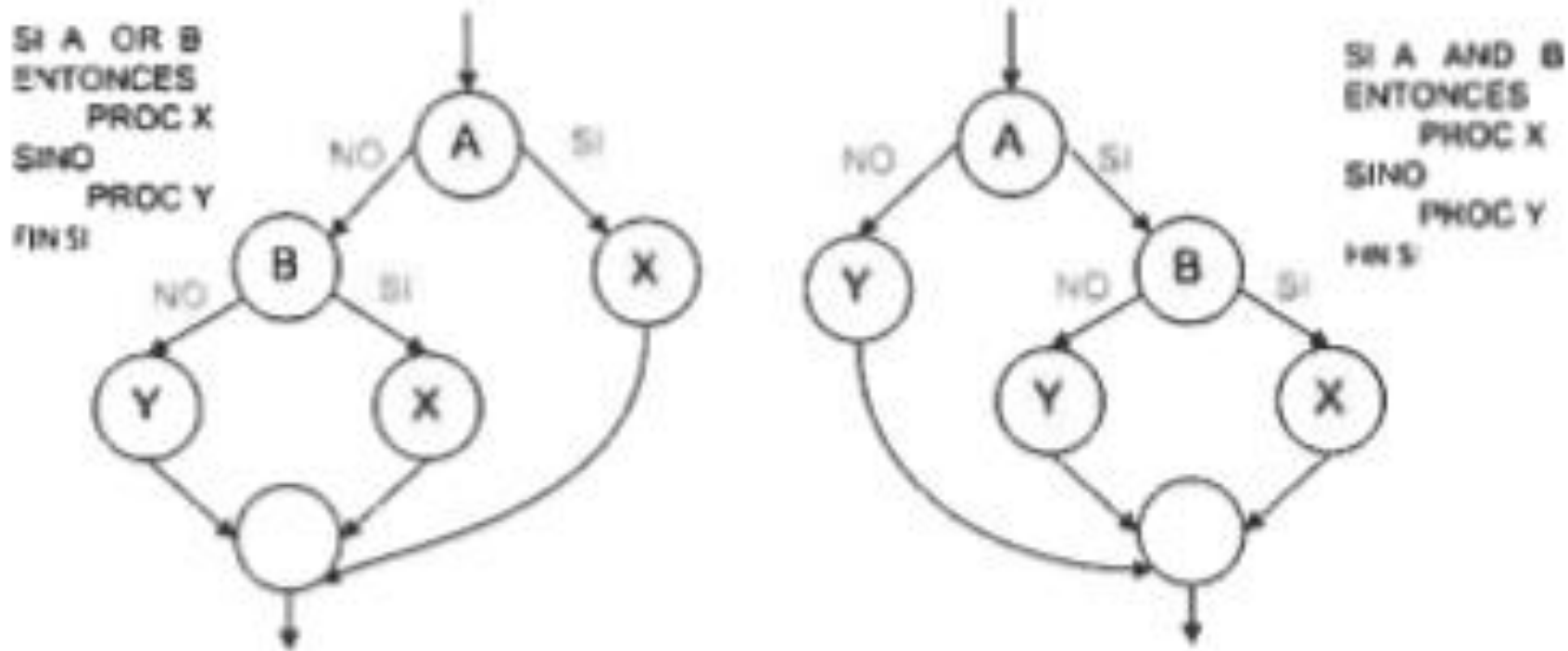


Figura 3.7. Lógica compuesta.

5. Pruebas de código

5.1 Prueba del camino básico

- **COMPLEJIDAD CILOMÁTICA**
- Es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa.
- Establece **el número de caminos independientes** del conjunto básico de caminos de ejecución de un Programa, y por lo tanto, **el número de casos de prueba** que se deben ejecutar para asegurar que cada sentencia se ejecuta al menos una vez.
- Se representa por $V(G)$ y se puede calcular de tres formas:
 1. $V(G)$ = Número de regiones del grafo.
 2. $V(G)$ = Aristas — Nodos + 2.
 3. $V(G)$ = Nodos predicado + 1

5. Pruebas de código

5.1 Prueba del camino básico

Para el *Ejemplo 1*, la complejidad ciclomática es 3.

1. $V(G) = \text{Número de regiones del grafo} = 3.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3.$
3. $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$

Para el *Ejemplo 2*, la complejidad ciclomática es 5.

1. $V(G) = \text{Número de regiones del grafo} = 5.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 14 - 11 + 2 = 5$
3. $V(G) = \text{Nodos predicado} + 1 = 4 + 1 = 5$

5. Pruebas de código

5.1 Prueba del camino básico

Se establecen los siguientes valores de referencia de la complejidad ciclomática:

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.

5. Pruebas de código

5.1 Prueba del camino básico

- **COMPLEJIDAD CILOMÁTICA**
- El valor de $V(G)$ nos da el número de caminos independientes del conjunto básico de un programa.
- Un **camino independiente** es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición.
- 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.

5. Pruebas de código

5.1 Prueba del camino básico

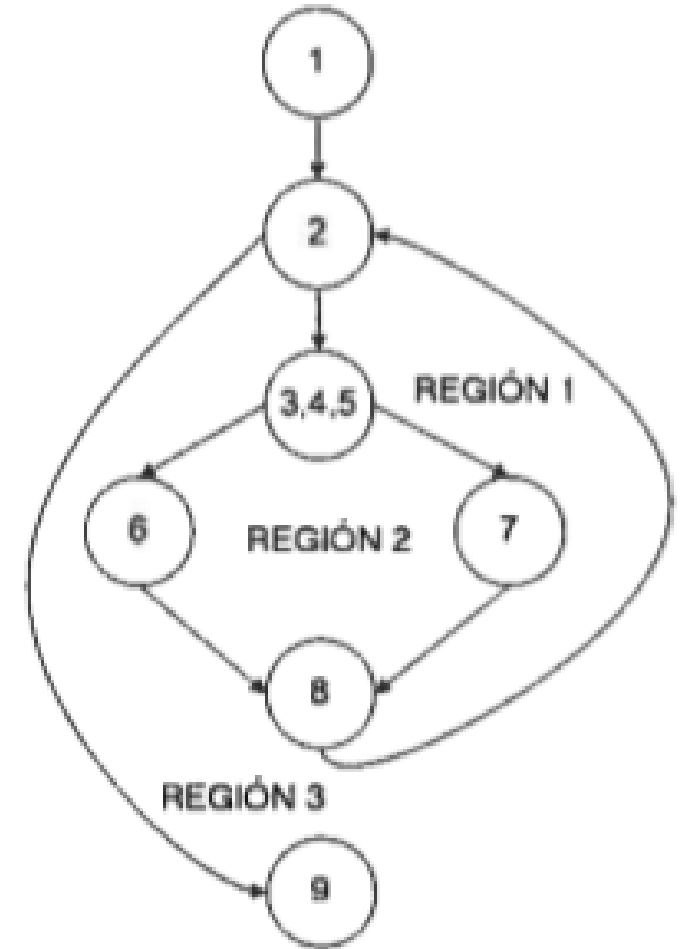
- COMPLEJIDAD CILOMÁTICA

Para el *Ejemplo 1*, la complejidad ciclomática es 3.

1. $V(G) = \text{Número de regiones del grafo} = 3.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3.$
3. $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$

Para el *Ejemplo 1*, un conjunto de caminos independientes será:

- Camino 1: 1 – 2 – 9
- Camino 2: 1 – 2 – 3, 4, 5 – 6 – 8 – 2 – 9
- Camino 3: 1 – 2 – 3, 4, 5 – 7 – 8 – 2 – 9



5. Pruebas de código

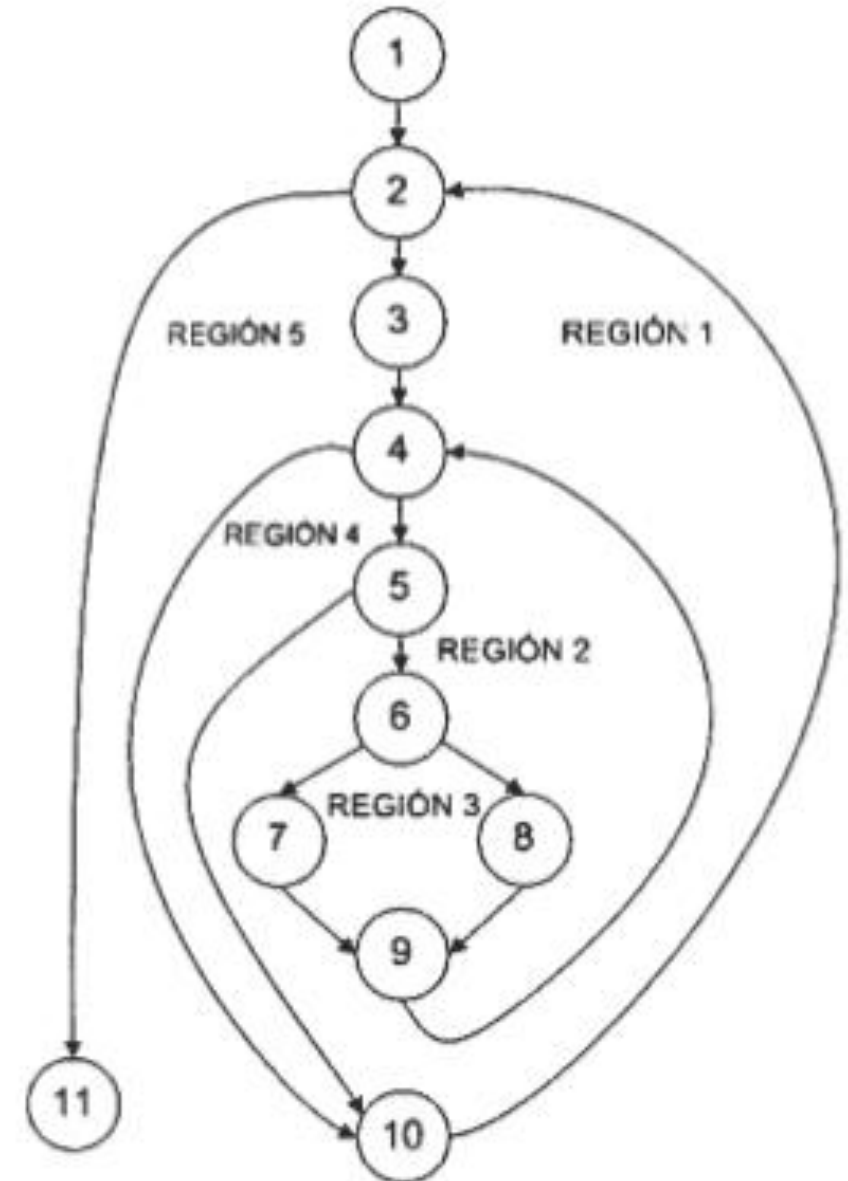
5.1 Prueba del camino básico

Para el *Ejemplo 2*, la complejidad ciclomática es 5.

1. $V(G) = \text{Número de regiones del grafo} = 5.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 14 - 11 + 2 = 5$
3. $V(G) = \text{Nodos predicado} + 1 = 4 + 1 = 5$

Para el *Ejemplo 2*, un conjunto de caminos independientes será:

- Camino 1: 1 – 2 – 11
- Camino 2: 1 – 2 – 3 – 4 – 10 – 2 – 11
- Camino 3: 1 – 2 – 3 – 4 – 5 – 10 – 2 – 11
- Camino 4: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 4 – 10 – 2 – 11
- Camino 5: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 9 – 4 – 10 – 2 – 11



5. Pruebas de código

5.1 Prueba del camino básico

- **OBTENCIÓN DE CASOS DE PRUEBA**

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

Camino 1: 1 - 2 - 9

Camino 2: 1 - 2 - 3, 4, 5 - 6 - 8 - 2 - 9

Camino 3: 1 - 2 - 3, 4, 5 - 7 - 8 - 2 - 9

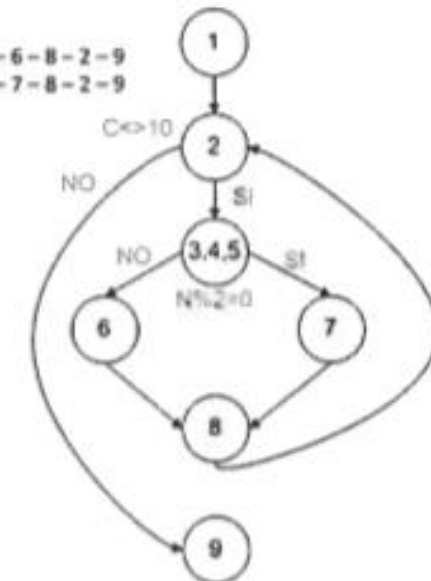


Figura 3.8. Nodos predichado del Ejemplo 1 con sus condiciones.

5. Pruebas de código

5.1 Prueba del camino básico

- **EJERCICIOS:** 1, 4 Y 5 página 143 y 144
- **Ejercicio de ampliación** 1 página 146

5. Pruebas de código

5.2 Partición o clases de equivalencia

- La **partición equivalente** es un método de prueba de caja negra que divide los valores de los campos de entrada de un programa en clases de equivalencia.
- Para identificar las clases de equivalencia se examina cada condición de entrada y se divide en dos o más grupos.
- Se definen dos tipos de clases de equivalencia:
 - **Clases válidas:** son los valores de entrada válidos.
 - **Clases no válidas:** son los valores de entrada no válidos.

5. Pruebas de código

5.2 Partición o clases de equivalencia

- Las clases de equivalencia se definen según una serie de directrices:

Condiciones de entrada	Nº de Clases de equivalencia válidas	Nº de Clases de equivalencia no válidas
1 . Rango	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

5. Pruebas de código

5.2 Partición o clases de equivalencia

Ejemplo 4: se va a realizar una entrada de datos de un empleado por pantalla gráfica, se definen 3 campos de entrada y una lista para elegir el oficio. La aplicación acepta los datos de esta manera:

- *Empleado*: número de tres dígitos que no empieza por 0.
- *Departamento*: en blanco o número de dos dígitos.
- *Oficio*: Analista, Diseñador, Programador o Elige oficio.

Si la entrada es correcta el programa asigna un salario (que se muestra en pantalla) a cada empleado 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 2000.

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*.

5. Pruebas de código

5.2 Partición o clases de equivalencia

Condición de entrada	Clases de equivalencia	Clases Válidas	COD	Clases no Válidas	COD
Empleado	Rango	100>= Empleado<=999	V1	Empleado < 100 Empleado >999	NV1 NV2
Departamento	Lógica (puede estar o no)	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 oficio"	NV8
		Oficio ="Analista"	V5		
		Oficio ="Diseñador"	V6		

CASO DE PRUEBA	Clases de 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, NV8	300		Elige oficio	ER3
CP8	V1, NV4, V6	345	123	Diseñador	ER2

5. Pruebas de código

5.2 Partición o clases de equivalencia

Ejemplo 5: tenemos una función Java que recibe un número entero y devuelve una cadena con el texto “Par” si el número recibido es par, o “Impar” si el número es impar.

```
public String parImpar(int nume) {  
    String cad="";  
    if(nume % 2 == 0)  
        cad="Par";  
    else  
        cad="Impar";  
    return cad;  
}
```

Condición de entrada	Clases de equivalencia	Clases Válidas	COD	Clases no Válidas	COD
nume	Valor Par	Cualquier número entero par	V7	Número impar Cadena	NV9 NV10
	Valor Impar	Cualquier número entero impar	V8	Número par Cadena	NV11 NV12

Los casos de prueba serían los siguientes:

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA	Resultado esperado
		nume	
CP1	V7	20	Par
CP2	V8	25	Impar
CP3	NV9	45	Error, número impar
CP4	NV10	“we”	Error, es una cadena
CP5	NV11	10	Error, número par
CP6	NV12	“ad”	Error, es una cadena

5. Pruebas de código

5.3 Análisis de valores límite

- El **análisis de valores límite** se basa en que los errores tienden a producirse con más probabilidad en los límites o extremos de los campos de entrada.
- Esta técnica complementa a la anterior y los casos de prueba elegidos ejercitan los valores justo por encima y por debajo de los márgenes de la clase de equivalencia.
- Además no solo se centra en las condiciones de entrada, sino que también se exploran las condiciones de salida definiendo las clases de equivalencia de salida.

5. Pruebas de código

5.3 Análisis de valores límite

- Las reglas son las siguientes:
 1. **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 valores enteros comprendidos entre 1 y 10, hay que escribir casos de prueba para el valor 1, 10, 0 y 11.
 2. **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 debajo del mínimo.** Por ejemplo, si el programa requiere de dos a diez datos de entradas, hay que escribir casos de prueba pal-a 2, 10. 1 y 11 datos de entrada.
 3. **Aplicar la regla 1 para la condición de salida.** Por ejemplo, si se debe aplicar sobre un campo de salida un descuento de entre un 10% mínimo y un 50% máximo (dependiendo del tipo de cliente); se generarán casos de prueba para 9,99%, 10%, 50% y 50,01%.

5. Pruebas de código

5.3 Análisis de valores límite

- Las reglas son las siguientes:
 4. **Usar la regla 2 para la condición de salida.** Por ejemplo, si la salida de un programa es una tabla de temperaturas de 1 a 10 elementos, se deben diseñar casos de prueba para que la salida del programa produzca 0, 1, 10 y 11 elementos. Tanto en esta regla, como en la anterior, hay que tener en cuenta que no siempre se podrán generar resultados fuera del rango de salida.
 5. **Si las estructuras de datos internas tienen límites preestablecidos** (por ejemplo un array de 100 elementos), **hay que asegurarse de diseñar casos de prueba que ejercite la estructura de datos en sus límites, primer y último elemento.**

5. Pruebas de código

5.3 Análisis de valores límite

Ejemplo 6: determina los casos de prueba para los siguientes elementos según las condiciones de entrada y de salida:

	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.

5. Pruebas de código

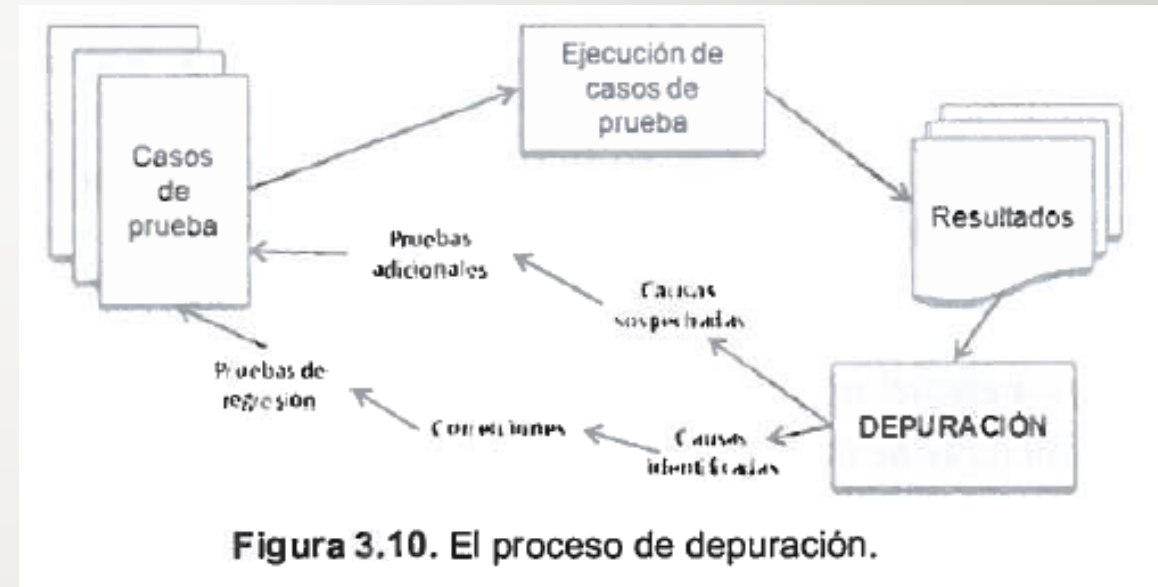
5.3 Análisis de valores límite

Ejemplo 7: partimos del *Empleado* (que tiene que ser un número de tres dígitos que no empiece por 0) del ejemplo 4 del epígrafe anterior. Utilizando esta técnica, para la clase de equivalencia *V1* que representa un rango de valores ($100 \geq \text{Empleado} \leq 999$) se deben generar dos casos de prueba con el límite inferior y el superior del rango (para identificar estos casos de prueba utilizamos *V1a* para el límite inferior y *V1b* para el superior):

CASO DE PRUEBA	Clases de 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	NV1, V3, V6	99	30	"Diseñador"	ER1
CP14	NV2, V2, V4	1000		"Programador"	ER1

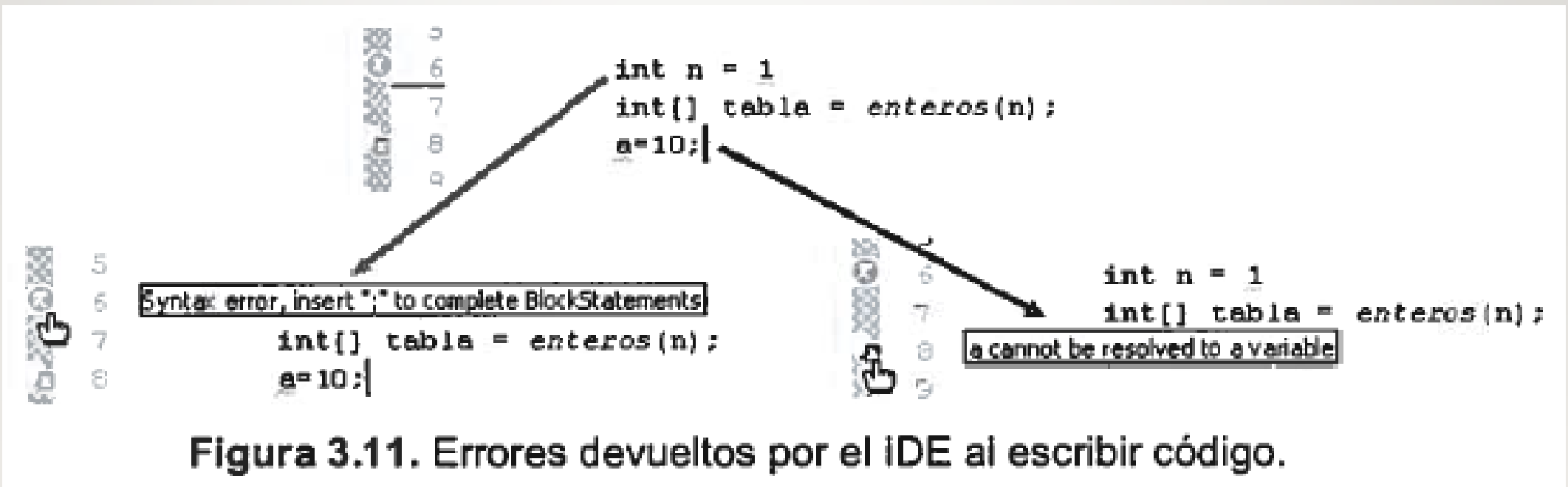
6. Herramientas de depuración

- 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 una falta de correspondencia entre los resultados esperados y los obtenidos realmente.
- El proceso de depuración siempre tiene uno de los dos resultados siguientes:
 - a) Se encuentra la causa del error, se corrige y se elimina
 - b) No se encuentra la causa del error, pero hay sospechas.



6. Herramientas de depuración

- Al desarrollar programas cometemos dos tipos de errores: errores de compilación y errores lógicos.



6. Herramientas de depuración

- En **Eclipse** podemos lanzar el depurador :
 - Pulsando en el botón Debug
 - Seleccionando el menú Run->Debug
 - Mediante el menú contextual que se muestra al hacer clic con el botón derecho del ratón en la clase que se va a ejecutar y seleccionando Debug .as -> Java Application.
- En cualquiera de esos casos la clase se ejecuta.

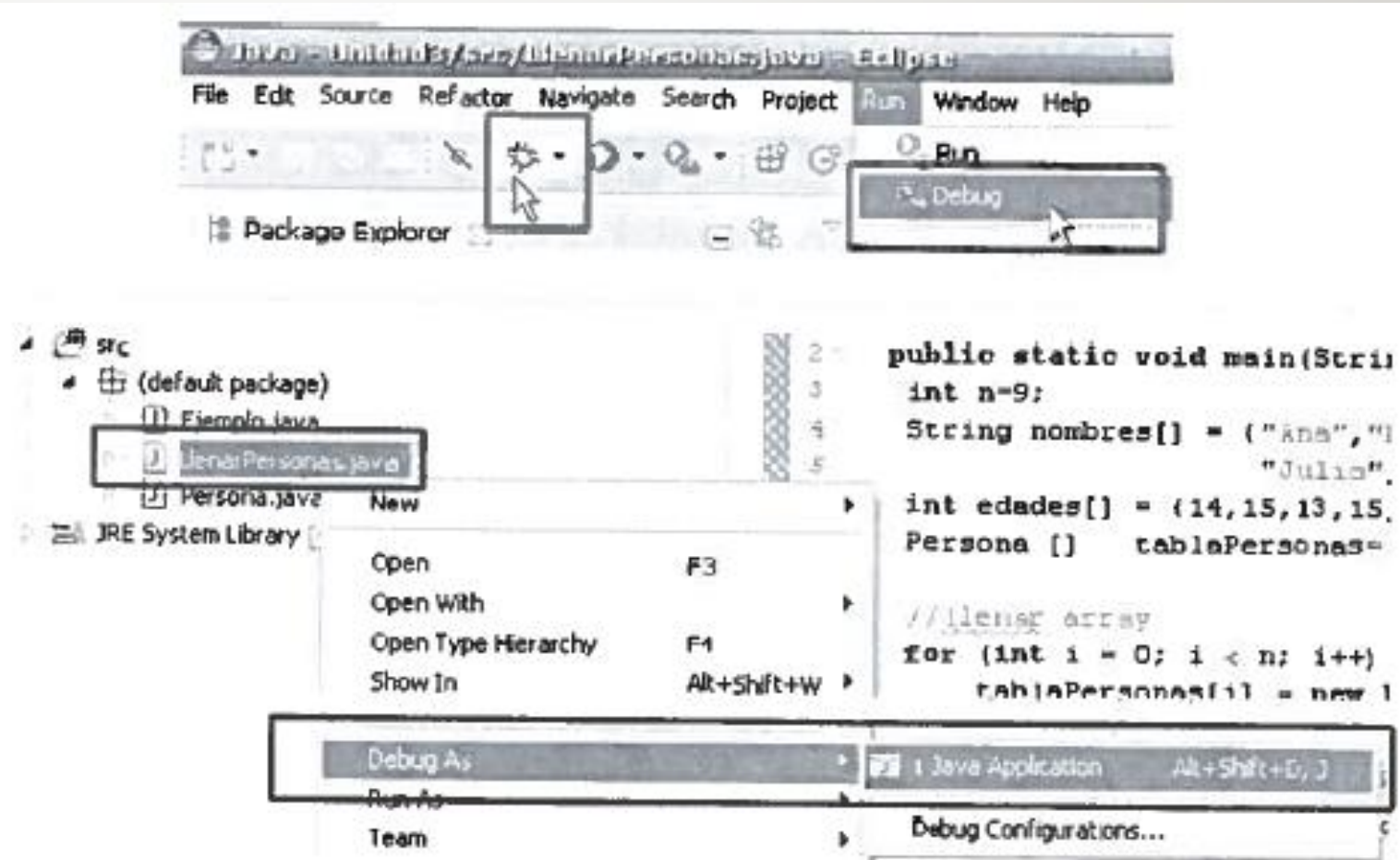
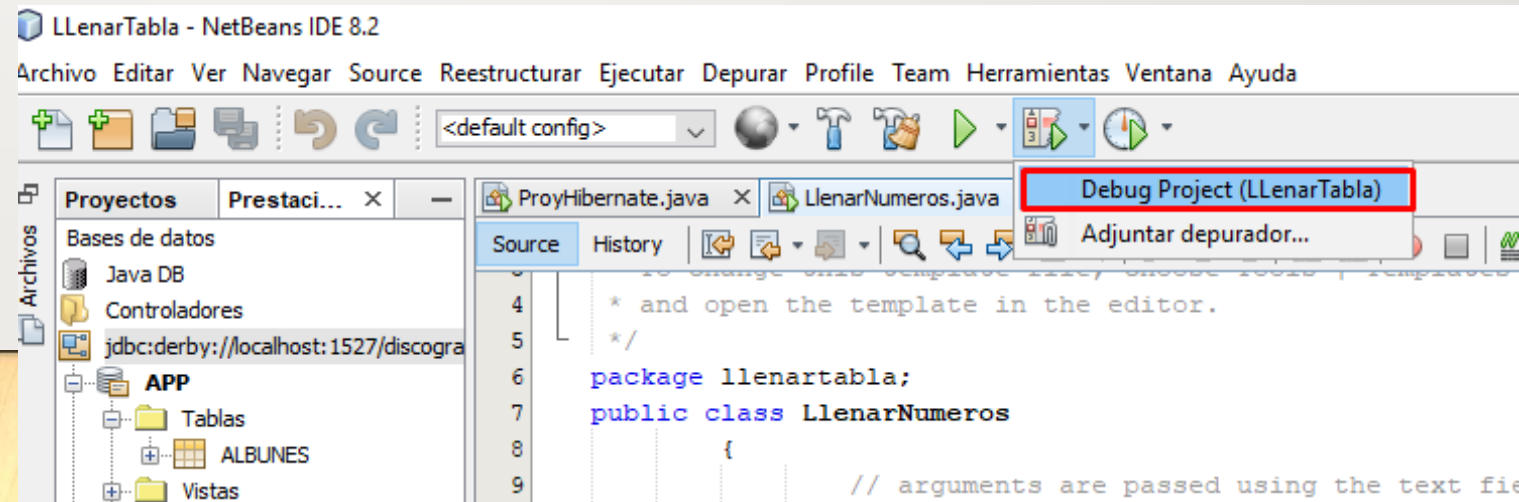
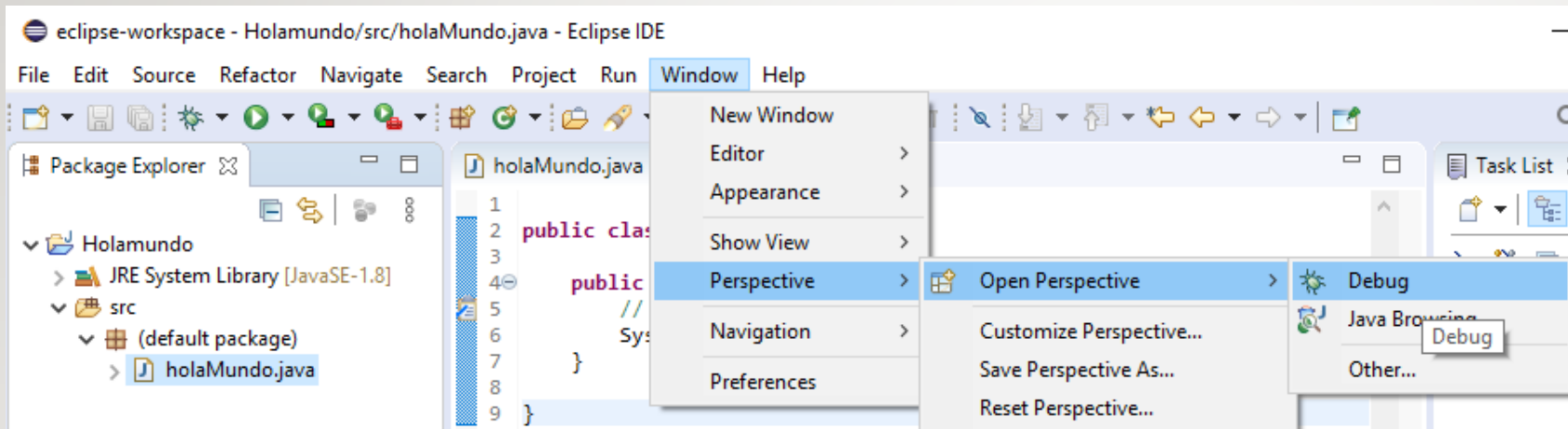


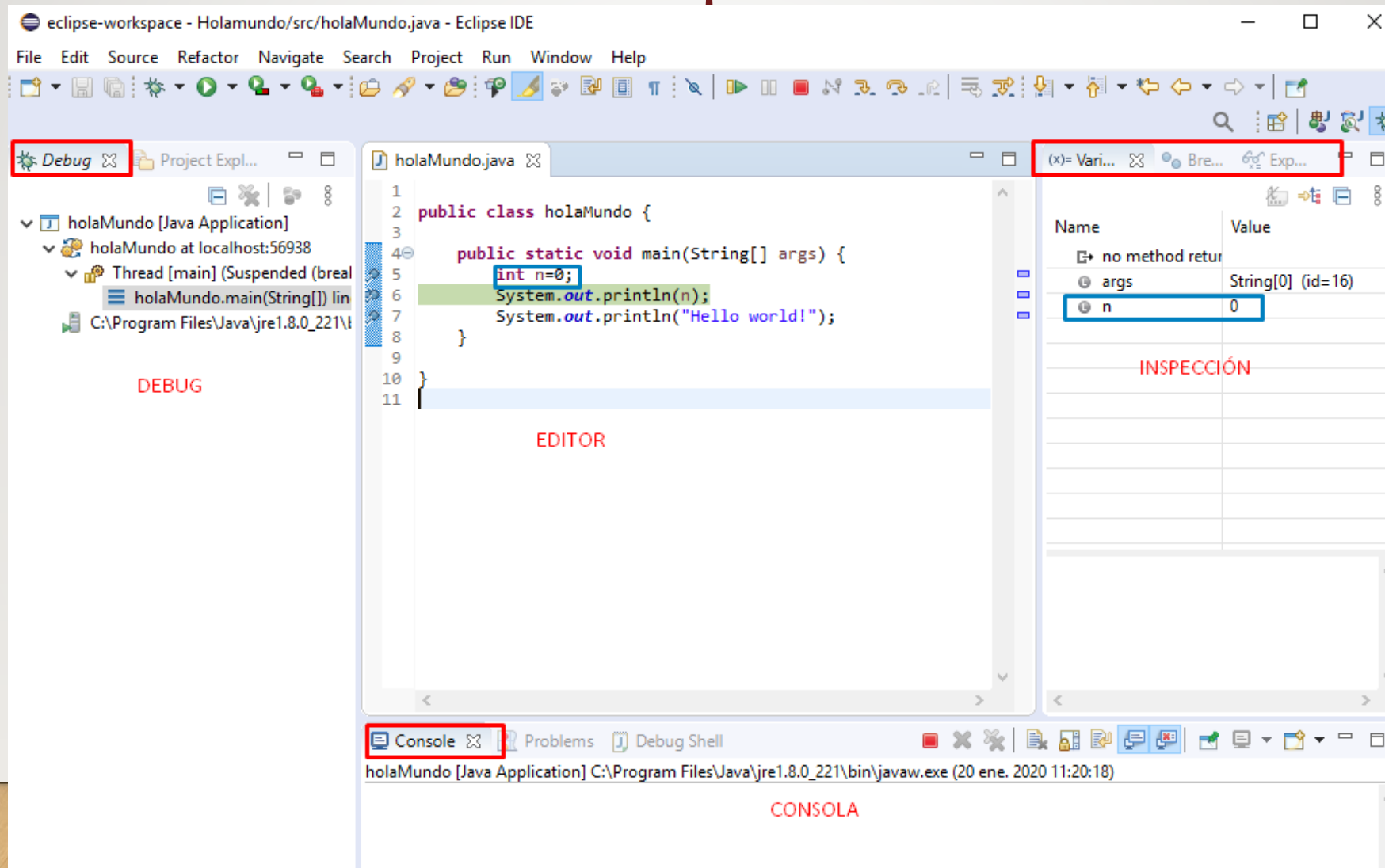
Figura 3.12. Lanzar el depurador en Eclipse.

6. Herramientas de depuración

- También es conveniente abrir la vista de depuración

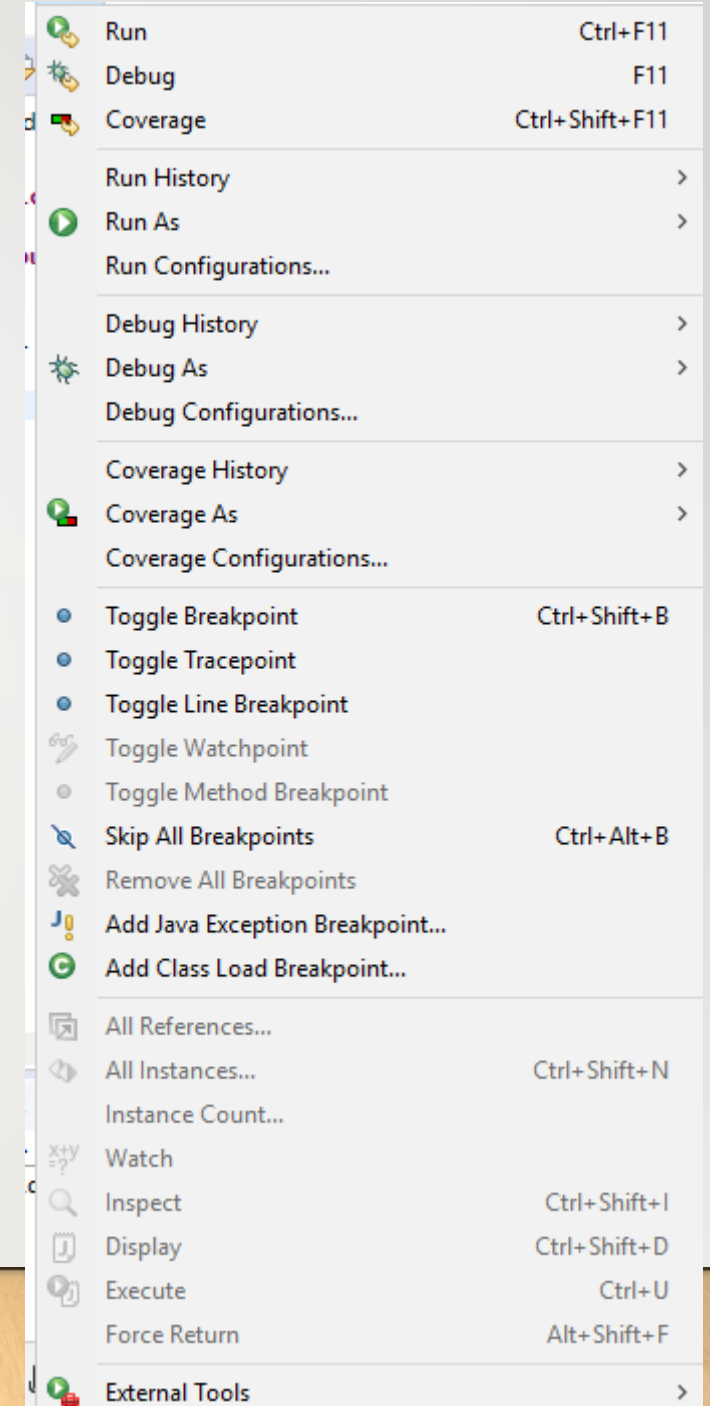
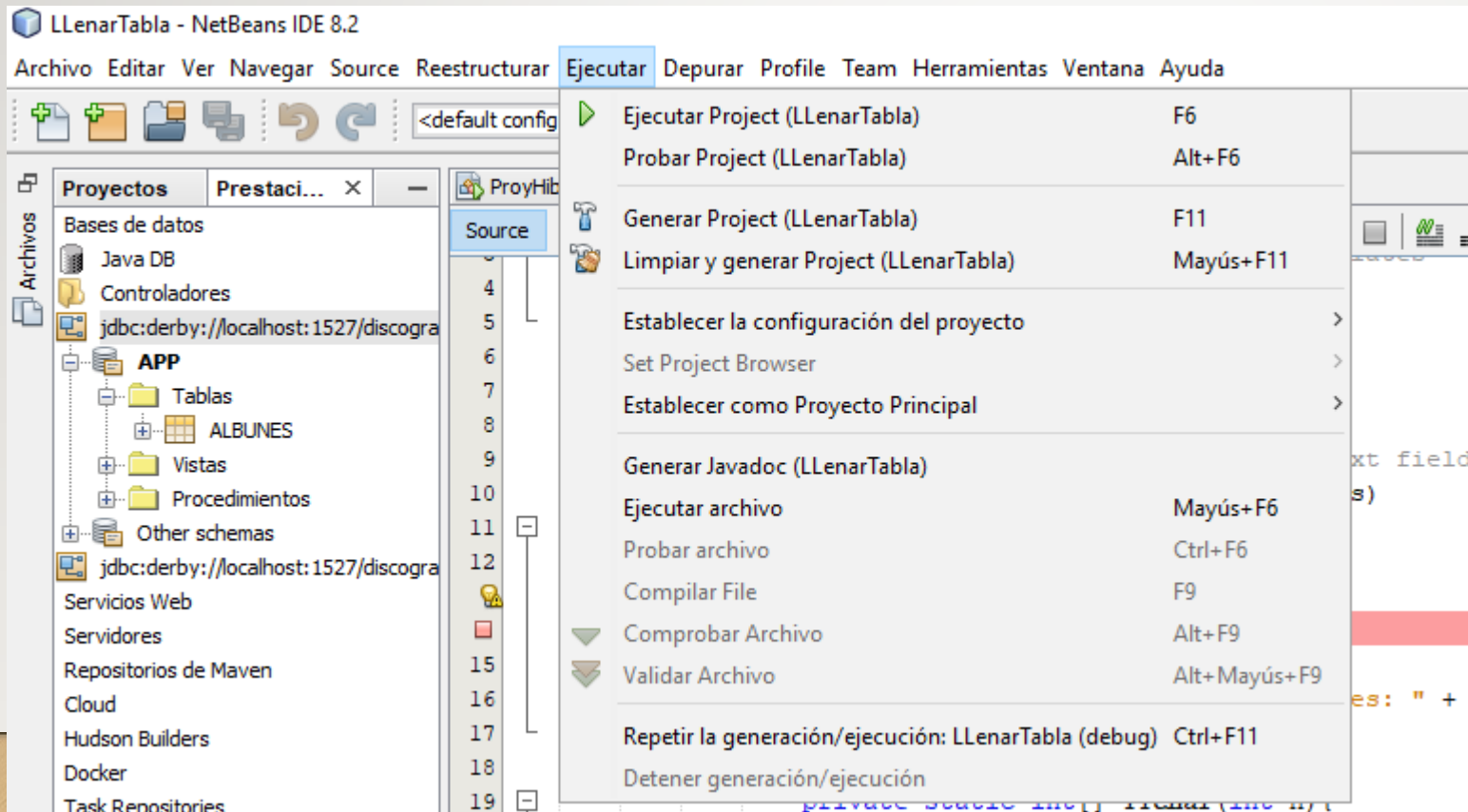


6. Herramientas de depuración



6. Herramientas de depuración

- Desde el **menu Run** de la perspectiva de depuración



6. Herramientas de depuración

6.1 Puntos de Ruptura y seguimiento

- Ejemplo puntos de ruptura

```
public class LlenarNumeros
{
    // arguments are passed using the text field below this editor
    public static void main(String[] args)
    {
        int n = 5;
        int[] tabla = new int[n];
        tabla = llenar(n);
        int suma = sumar (tabla);
        System.out.println("La suma es: + suma");
    }

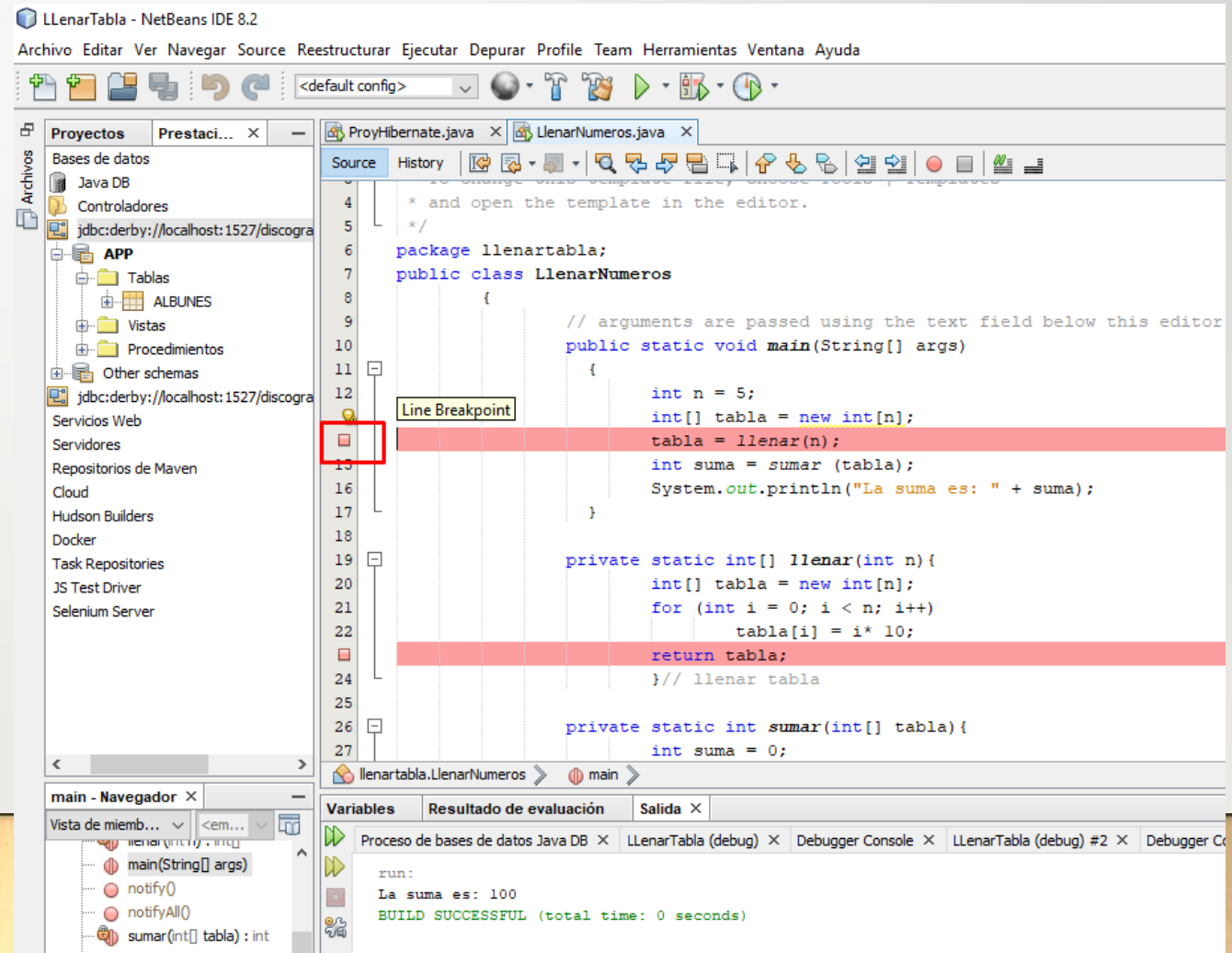
    private static int[] llenar(int n){
        int[] tabla = new int[n];
        for (int i = 0; i < n; i++)
            tabla[i] = i* 10;
        return tabla;
    } // llenar tabla

    private static int sumar(int[] tabla)
    {
        int suma = 0;
        int ri=tabla.lenght;
        for (int i = 0; i < n; i++)
            suma = suma + tabla[i];
        return suma;
    } //sumar tabla
} //fin clase
```

6. Herramientas de depuración

6.1 Puntos de Ruptura y seguimiento

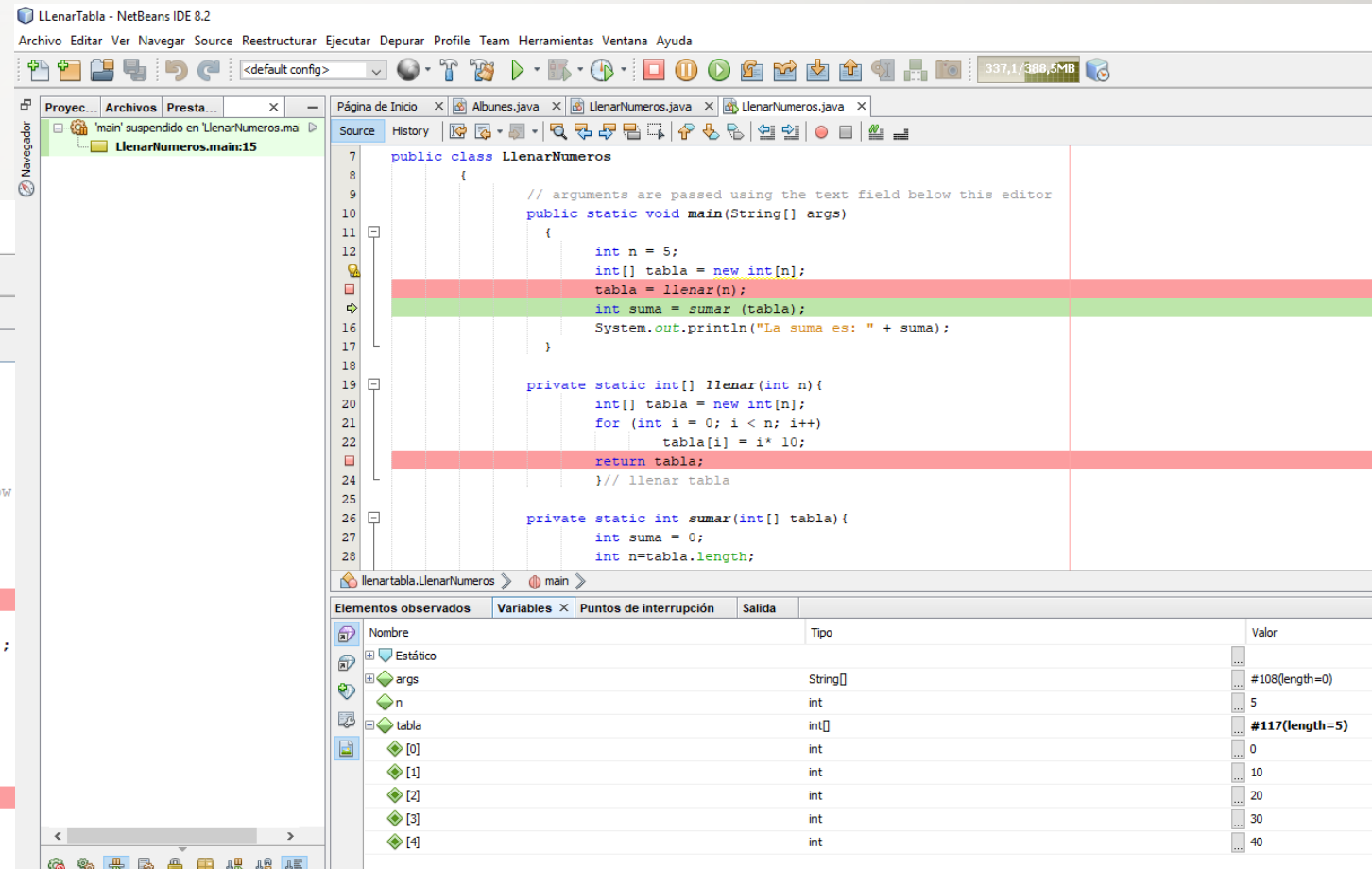
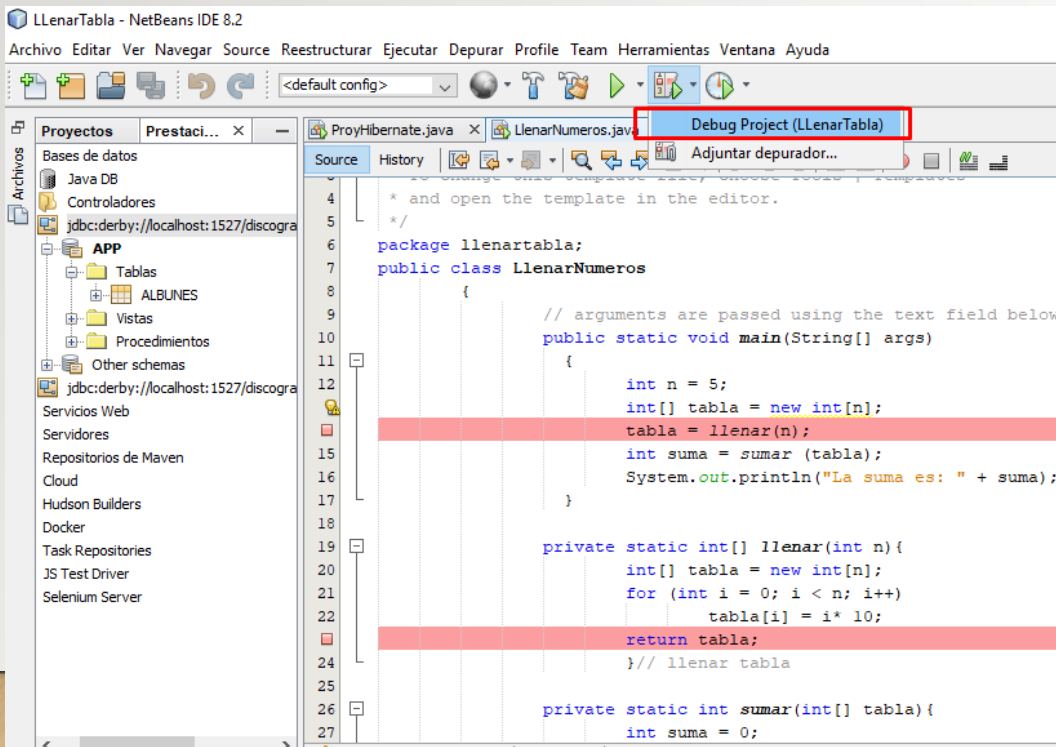
- Ejemplo puntos de ruptura
- Netbeans



6. Herramientas de depuración

6.1 Puntos de Ruptura y seguimiento

- Ejemplo puntos de ruptura
- Netbeans



7. JUnit

6.1 Creación de una clase de prueba

```
public class Calculadora {  
    private int num1;  
    private int num2;  
  
    public Calculadora(int a, int b) {  
        num1 = a;  
        num2 = b;  
    }  
    public int suma() {  
        int resul = num1 + num2;  
        return resul;  
    }  
    public int resta() {  
        int resul = num1 - num2;  
        return resul;  
    }  
    public int multiplica() {  
        int resul = num1 * num2;  
        return resul;  
    }  
    public int divide() {  
        int resul = num1 / num2;  
        return resul;  
    }  
}
```

7. JUnit

7.2 Preparación de los casos de prueba

MÉTODOS	MISIÓN
<code>assertTrue(boolean expresión)</code> <code>assertTrue(String mensaje, boolean expresión)</code>	Comprueba que la expresión se evalúe a <i>true</i> . Si no es <i>true</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertFalse(boolean expresión)</code> <code>assertFalse(String mensaje, boolean expresión)</code>	Comprueba que la expresión se evalúe a <i>false</i> . Si no es <i>false</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertEquals(valorEsperado,valorReal),</code> <code>assertEquals(String mensaje, valorEsperado,valorReal)</code>	Comprueba que el <i>valorEsperado</i> sea igual al <i>valorReal</i> . Si no son iguales y se incluye el String, entonces se lanzará el <i>mensaje</i> . <i>ValorEsperado</i> y <i>valorReal</i> pueden ser de diferentes tipos
<code>assertNull(Object objeto),</code> <code>assertNull(String mensaje, Object objeto)</code>	Comprueba que el <i>objeto</i> sea null. Si no es null y se incluye el String al producirse error se lanzará el <i>mensaje</i> .
<code>assertNotNull(Object objeto),</code> <code>assertNotNull(String mensaje, Object objeto)</code>	Comprueba que el <i>objeto</i> no sea null. Si es null y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertSame(Object objetoEsperado, Object objetoReal)</code> <code>assertSame(String mensaje, Object objetoEsperado, Object objetoReal)</code>	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertNotSame(Object objetoEsperado, Object objetoReal)</code> <code>assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal)</code>	Comprueba que <i>objetoEsperado</i> no sea el mismo objeto que <i>objetoReal</i> . Si son el mismo y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>fail()</code> <code>fail(String mensaje):</code>	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el <i>mensaje</i>

7. JUnit

7.2 Preparación de los casos de prueba

- Lanzar excepciones

```
public int divide0() {  
    if(num2==0)  
        throw new java.lang.ArithmeticException("División por 0");  
    else{  
        int resul= num1 / num2;  
        return resul;  
    }  
}
```

```
@Test(expected = java.lang.ArithmeticException.class)  
public void testDivide0() {  
    Calculadora calculo = new Calculadora(20, 0);  
    Integer resultado = calculo.divide0();  
}
```


7. JUnit

7.3 Tipos de anotaciones

@Before: si anotamos un método con esta etiqueta, el código será ejecutado antes de cualquier método de prueba. Este método se puede utilizar para inicializar datos, por ejemplo, en una aplicación de acceso a base de datos se puede preparar la base de datos, si vamos a usar un array para las pruebas se puede inicializar aquí. Puede haber varios métodos en la clase de prueba con esta anotación.

@After: si anotamos un método con esta etiqueta el código será ejecutado después de la ejecución de todos los métodos de prueba. Se puede utilizar para limpiar datos. Puede haber varios métodos en la clase de prueba con esta anotación.

```
public class CalculadoraTest2 {  
    private Calculadora calculo;  
    private int resultado;  
  
    @Before  
    public void creaCalculadora() {  
        calculo = new Calculadora(20, 10);  
    }  
  
    @After  
    public void borraCalculadora() {  
        calculo = null;  
    }  
  
    @Test  
    public void testSuma() {
```


7. JUnit

7.3 Tipos de anotaciones

@BeforeClass: solo puede haber un método con esta etiqueta. El método marcado con esta anotación es invocado una vez al principio del lanzamiento de todas las pruebas. Se suele utilizar para inicializar atributos comunes a todas las pruebas o para realizar acciones que tardan un tiempo considerable en ejecutarse.

@AfterClass: solo puede haber un método con esta anotación. Este método será invocado una sola vez cuando finalicen todas las pruebas.

```
public class CalculadoraTest3 {  
    private static Calculadora calculo;  
    private int resultado;  
  
    @BeforeClass  
    public static void creaCalculadora() {  
        calculo = new Calculadora(20, 10);  
    }  
  
    @AfterClass  
    public static void borraCalculadora() {  
        calculo = null;  
    }  
}
```

7. JUnit

7.3 Pruebas parametrizadas

```
import static org.junit.Assert.*;
import java.util.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class CalculadoraTest4 {
    private int num1;
    private int num2;
    private int resul;

    public CalculadoraTest4(int num1, int num2, int resul) {
        this.num1 = num1;
        this.num2 = num2;
        this.resul = resul; //resultado de la operación
    }

    @Parameters
    public static Collection<Object[]> numeros() {
        return Arrays.asList(new Object[][] {
            { 20, 10, 2 }, { 30, -2, -15 }, { 5, 2, 3 }
        });
    }
}
```

```
    }

    @Test
    public void testDivide() {
        Calculadora calculo = new Calculadora(num1, num2);
        int resultado = calculo.divide();
        assertEquals(resul, resultado);
    }
}
```

7. JUnit

7.3 Suite de Pruebas

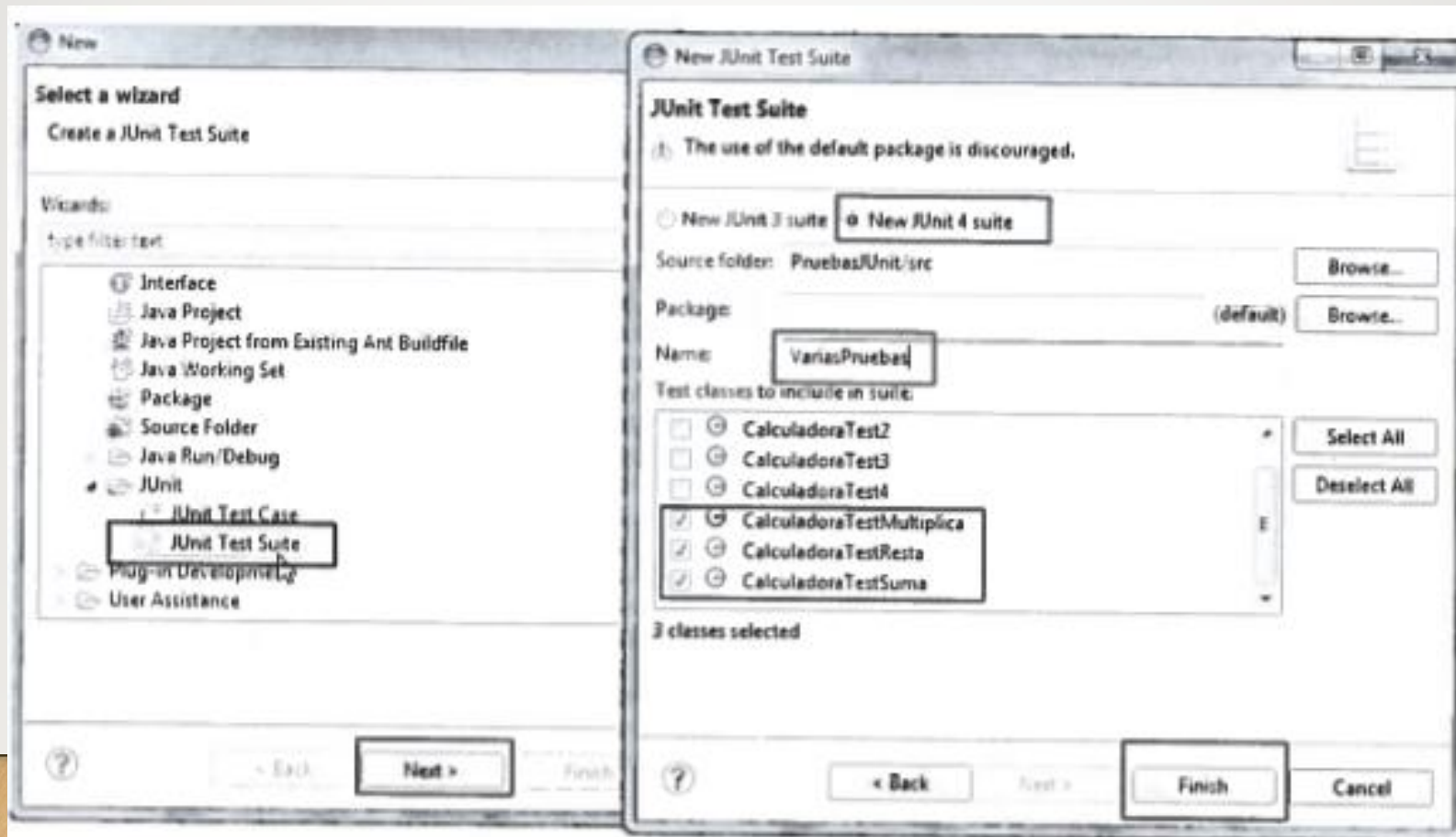


Figura 3.27. Creación de la suite de pruebas.

7. JUnit

7.3 Suite de Pruebas

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
    CalculadoraTestMultiplica.class,
    CalculadoraTestResta.class,
    CalculadoraTestSuma.class
})

public class VariasPruebas {

}
```