ENTORNOS DE DESARROLLO
TEMA 3: DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

CFGS - DAW 1

JESUITAS

Sagrado Corazón
Jesuitas · Logroño

Índice

1. Planificación de las pruebas

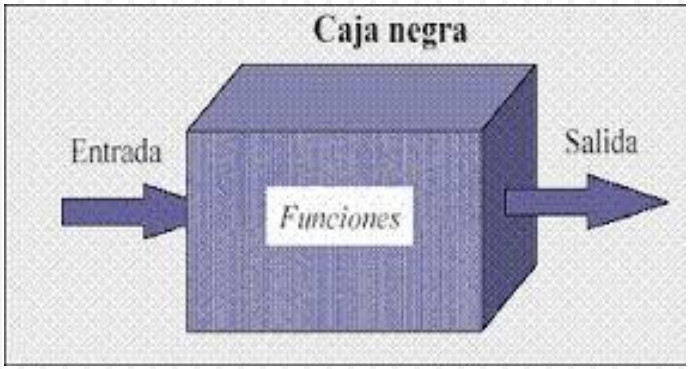
- Durante todo el proceso de desarrollo de software, desde la fase de diseño, en la implementación y una vez desarrollada la aplicación, es necesario realizar un conjunto de pruebas que permitan verificar que el software que se está creando, es correcto y cumple con las especificaciones impuesta por el usuario.

1. Planificación de las pruebas

- Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas a lo largo de toda la duración del proyecto:
  1. Las pruebas empezarían con las **pruebas unitarias** realizadas por cada desarrollador en el código implementado.
  2. Después estarían las **pruebas de integración**, donde se prestan atención al diseño y la construcción de la arquitectura del software.
  3. El siguiente paso sería **las pruebas de validación**, donde se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software.
  4. Finalmente, **pruebas de sistema** donde se verifica el funcionamiento total del software y otros elementos del sistema.

2. Tipos de pruebas

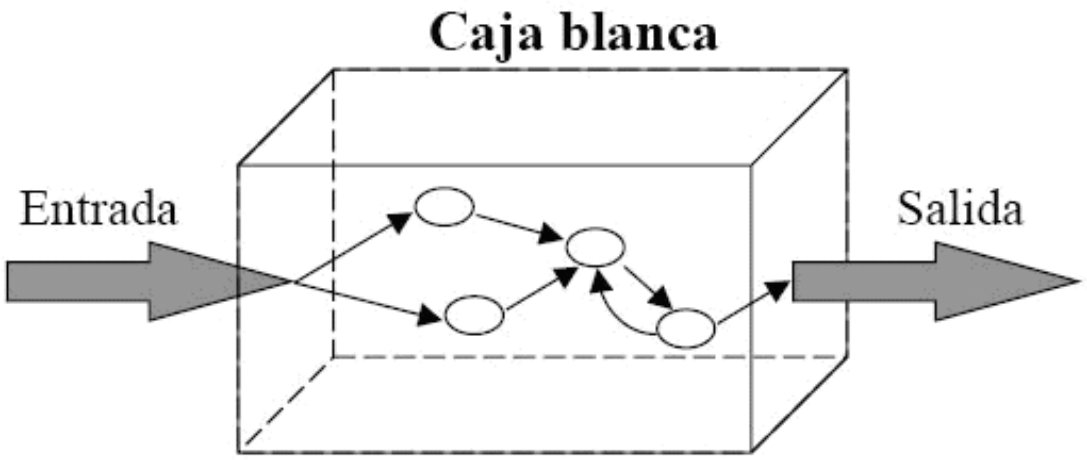
- Dos enfoques fundamentales:
  A. **Prueba de la Caja Negra (Black Box Testing):**
    - Lo fundamental es comprobar que los resultados de la ejecución de la aplicación, son los esperados, en función de las entradas que recibe.
    - Se lleva a cabo sin tener que conocer ni la estructura, ni el funcionamiento 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.



2. Tipos de pruebas

B. **Prueba de la Caja Blanca (White Box Testing):**

- En este caso, se prueba la aplicación desde dentro, usando su lógica de aplicación.
- Se analiza y prueba directamente el código de la aplicación.
- Es necesario un conocimiento específico del código, para poder analizar los resultados de las pruebas.



2.1. Pruebas de caja negra

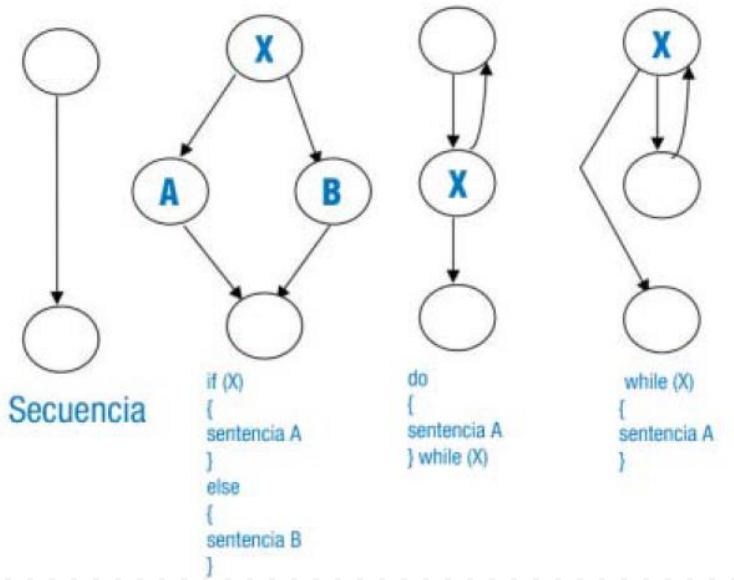
- También llamadas funcionales.
- Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos
- No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.
- Las pruebas funcionales intentarían responder a las preguntas ¿puede el usuario hacer esto? o ¿funciona esta utilidad de la aplicación?
  - Ejemplo:
    - *Si estamos implementando una aplicación que realiza un determinado cálculo científico, en el enfoque de las pruebas funcionales, solo nos interesa verificar que ante una determinada entrada a ese programa el resultado de la ejecución del mismo devuelve como resultado los datos esperados.*
    - *No se consideraría, en ningún caso, el código desarrollado, ni su eficiencia, ni el algoritmo empleado, ni si hay parte de código innecesarias,etc.*

2.1. Pruebas de caja negra

- Tipos:
  a) Partición equivalente: Consiste en dividir y separar los campos de entrada según el tipo de datos y las restricciones que conllevan. De esta forma se definen unas pruebas comunes dependiendo del tipo de dato, asegurando así pruebas eficaces y completas.
     - Ejemplo: Campo *codigoPostal*, que no es obligatorio rellenar, pero que solo admite entradas de 5 dígitos numéricos → Habrá que valorar si está o no está presente, en caso de estar presente deberíamos confirmar que se trata de un número de 5 cifras con un rango mínimo y máximo (10000-99999) o también que la longitud sea 5 y solo sean números.
  b) Análisis de valores límites: En este caso, a la hora de implementar un caso de prueba, se van a elegir como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia.
  c) Pruebas aleatorias: Consiste en generar entradas aleatorias para la aplicación que hay que probar. Se suelen utilizar generadores de prueba, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación.

2.2. Pruebas de caja blanca

- También llamadas estructurales.
- Para ver cómo el programa se va ejecutando, y así comprobar su correcta ejecución, se utilizan las pruebas estructurales, que se fijan en los caminos que se pueden recorrer.
- Pretenden comprobar que:
  - Se van a ejecutar todas la instrucciones del programa
  - Que no hay código no usado
  - Que los caminos lógicos del programa se van a recorrer



2.2. Pruebas de caja blanca

- Existen diferentes criterios:
  - **Cobertura de decisiones:** se trata de crear los suficientes casos de prueba para que cada opción resultado de una decisión se evalúe al menos una vez a cierto y otra a falso.
  - **Cobertura de condiciones:** se trata de crear los suficientes casos de prueba para que cada elemento de una condición, se evalúe al menos una vez a falso y otra a verdadero.
  - **Cobertura de condiciones y decisiones:** consiste en cumplir simultáneamente las dos anteriores.
  - **Cobertura de caminos:** es el criterio más importante. Establece que se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final. La ejecución de este conjunto de sentencias, se conoce como camino. Como el número de caminos que puede tener una aplicación, puede ser muy grande, para realizar esta prueba, se reduce el número a lo que se conoce como camino prueba.

2.2. Pruebas de caja blanca

- Ejemplo para comprobar que todas las funciones, sentencias, decisiones, y condiciones, se van a ejecutar:
  - Si durante la ejecución del programa, la función es llamada, al menos una vez, el cubrimiento de la función es satisfecho.
  - El **cubrimiento de sentencias** para esta función, será satisfecho si es invocada, por ejemplo como prueba(1,1), ya que en esta caso, cada línea de la función se ejecuta, incluida z=x;

```
int prueba (int x, int y)
{
        int z=0;
        if ((x>0) && (y>0))
        {
                z=x;
        }
        return z;
}
```

2.2. Pruebas de caja blanca

- Ejemplo (cont.):
  - Si invocamos a la función con prueba(1,1) y prueba(0,1), se satisfará el **cubrimiento de decisión**. En el primer caso, la if condición va a ser verdadera, se va a ejecutar z=x, pero en el segundo caso, no.
  - El **cubrimiento de condición** puede satisfacerse si probamos con prueba(1,1), prueba(1,0) y prueba(0,0). En los dos primeros casos (x>0) se evalúc a verdad mientras que en el tercero, se evalúa a falso. Al mismo tiempo, el primer caso hace (y>0) verdad, mientras el tercero lo hace falso.

```
int prueba (int x, int y)
{
        int z=0;
        if ((x>0) && (y>0))
        {
                z=x;
        }
        return z;
}
```

2.3. Pruebas de regresión

- Durante el proceso de prueba, tendremos éxito si detectamos un posible fallo o error.
- La consecuencia directa de ese descubrimiento, supone la modificación del componente donde se ha detectado.
- Esta modificación, puede generar errores colaterales, que no existían antes.
- Como consecuencia, la modificación realizada nos obliga a repetir pruebas que hemos realizado con anterioridad → Pruebas de regresión.

2.3. Pruebas de regresión

- Para ello, para que el número de pruebas no crezca demasiado:
  - Se selecciona una muestra de pruebas para probar la funcionalidad del software.
  - De forma más concreta, se realizan los casos de prueba asociados al las funcionalidades que se han cambiado.
  - También se prueban de forma más exhaustiva los componentes involucrados.

3. Herramientas de depuración

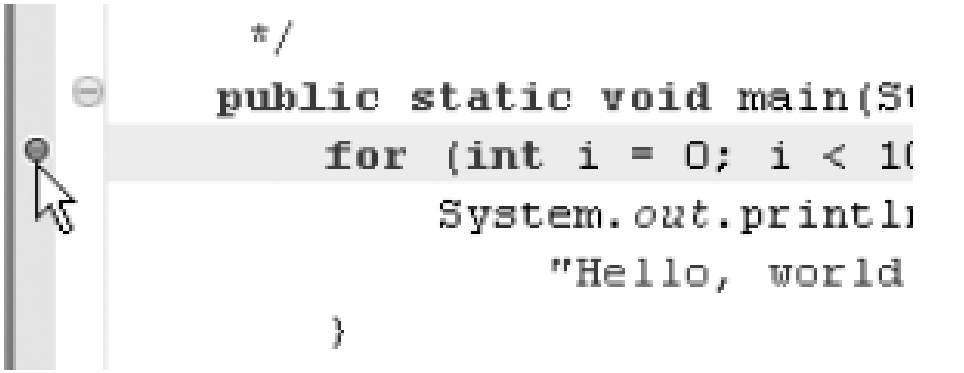
- Cada IDE incluye herramientas de depuración como:
  - Inclusión de puntos de ruptura
  - Ejecución paso a paso de cada instrucción
  - Ejecución por procedimiento
  - Inspección de variables, etc.
- Nos ayudan a encontrar errores lógicos, o bugs, que aun compilando correctamente el programa, provocan que el programa no funcione según lo esperado.
- Para ello, en los IDE → **depurador** o ***debugger***

3. Herramientas de depuración

- Debugger:
  - Un programa debe compilarse con éxito para poder utilizarlo en el depurador.
  - El depurador nos permite analizar todo el programa, mientras éste se ejecuta.
  - Permiten:
    - Suspender la ejecución de un programa
    - Examinar y establecer los valores de las variables
    - Comprobar los valores devueltos por un determinado método
    - Comprobar el resultado de una comparación lógica o relacional
    - Etc.

3.1. Puntos de ruptura

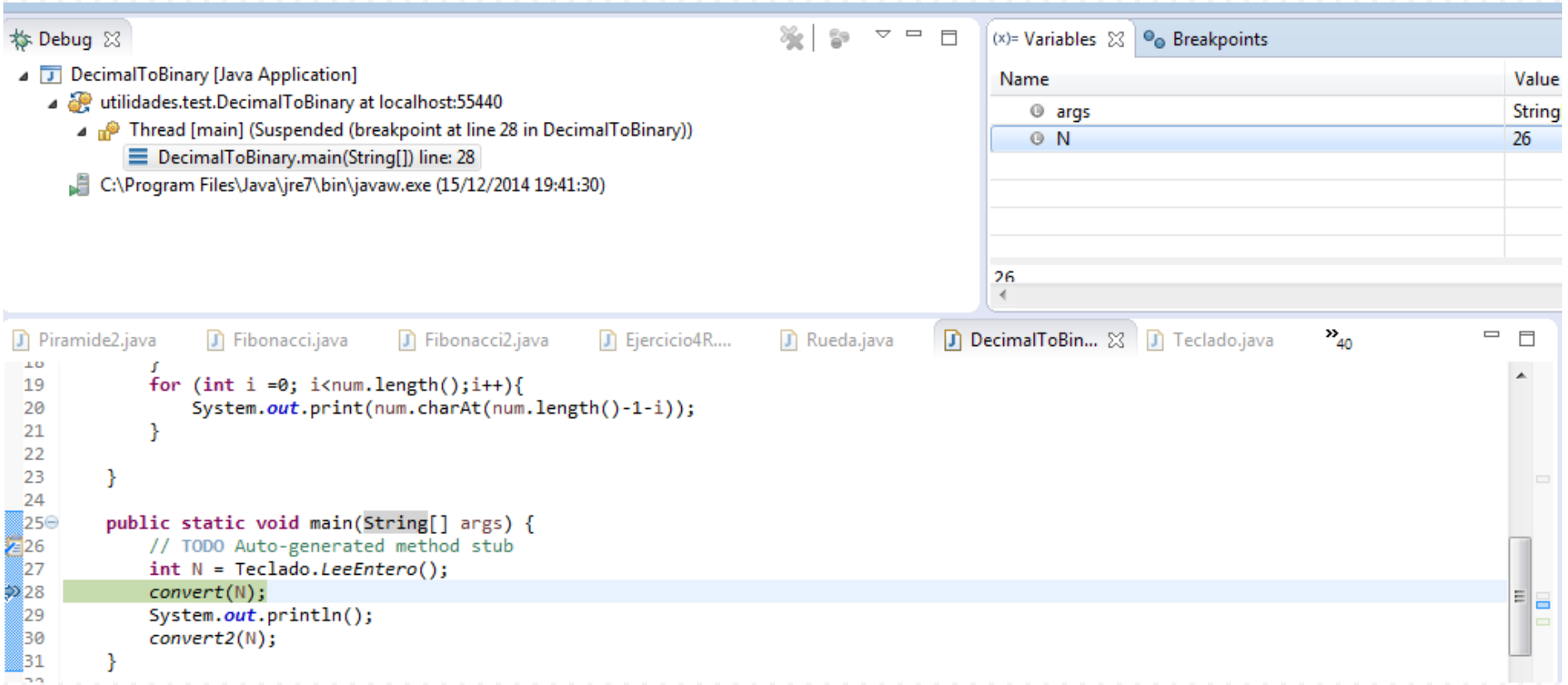
- Dentro de una aplicación, se pueden insertar varios puntos de ruptura, y se pueden eliminar con la misma facilidad con la que se insertan.
- Los puntos de ruptura son marcadores que pueden establecerse en cualquier línea de código ejecutable.



3.1. Puntos de ruptura

- 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:
  - Se pueden examinar las variables
  - Comprobar que los valores que tienen asignados son correctos
  - Se pueden iniciar una depuración paso a paso, e ir comprobando el camino que toma el programa a partir del punto de ruptura.

3.1. Puntos de ruptura



3.2. Tipos de ejecución

- Para poder depurar un programa, podemos ejecutar el programa de diferentes formas:
  A. 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.
  B. 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.

3.2. Tipos de ejecución

C. 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.

D. 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.

4. Normas de calidad

- Estándares BSI
  - BS 7925-1, Pruebas de software. Parte 1. Vocabulario.
  - BS 7925-2, Pruebas de software. Parte 2. Pruebas de los componentes software.
- Estándares IEEE de pruebas de software:
  - IEEE estándar 829, Documentación de la prueba de software.
  - IEEE estándar 1008, Pruebas de unidad
  - ISO / IEC 12207, 15289
  - La norma **ISO/IEC 29119** de prueba de software, pretende unificar en una única norma, todos los estándares, de forma que proporcione vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida del software.

5. Pruebas unitarias

- Las pruebas unitarias, tienen por objetivo probar el correcto funcionamiento de un módulo de código.
- El fin que se persigue, es que cada módulo funciona correctamente por separado.
- En programación orientada a objetos, una unidad es normalmente un método.

5. Pruebas unitarias

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

5. Pruebas unitarias

- Herramientas para Java: Jtiger, TestNG, **Junit**, etc.
- **JUnit** es una herramienta de automatización de pruebas que nos permite de manera rápida y sencilla, elaborar pruebas.
  - La herramienta nos permite diseñar clases de prueba, para cada clase diseñada en nuestra aplicación.
  - Una vez creada las clases de prueba, establecemos los métodos que queremos probar, y para ello diseñamos casos de prueba.
  - Una vez diseñados los casos de prueba, pasamos a probar la aplicación.
  - La herramienta de automatización nos presentará un informe con los resultados de la prueba.
  - En función de los resultados, deberemos o no, modificar el código.

5. Pruebas unitarias

A3.1 Pruebas unitarias mediante Junit (Moodle) (opcional)

B3.1 Pruebas unitarias mediante Junit de .net (Moodle)

E5.2. Entregable Casos de prueba con Junit (opcional)

E5.3. Entregable Casos de prueba con Junit de .net

6. Documentación de pruebas

- Los documentos que se generan son:
  - **Plan de Pruebas.**
    - **Especificación del diseño de pruebas.**
    - **Especificación de un caso de prueba.** Los casos de prueba se concretan a partir de la especificación del diseño de pruebas.
    - **Especificación de procedimiento de prueba.** Una vez especificado un caso de prueba, será preciso detallar el modo en que van a ser ejecutados cada uno de los casos de prueba.
  - **Registro de pruebas.** Se registrarán los sucesos que tengan lugar durante las pruebas.
  - **Informe de incidente de pruebas.** Para cada incidente, defecto detectado, solicitud de mejora, etc., se elaborará un "informe de incidente de pruebas".
  - **Informe sumario de pruebas.** Resumirá las actividades de prueba realizadas.