**Tema 6: Diseño y Realización de pruebas**

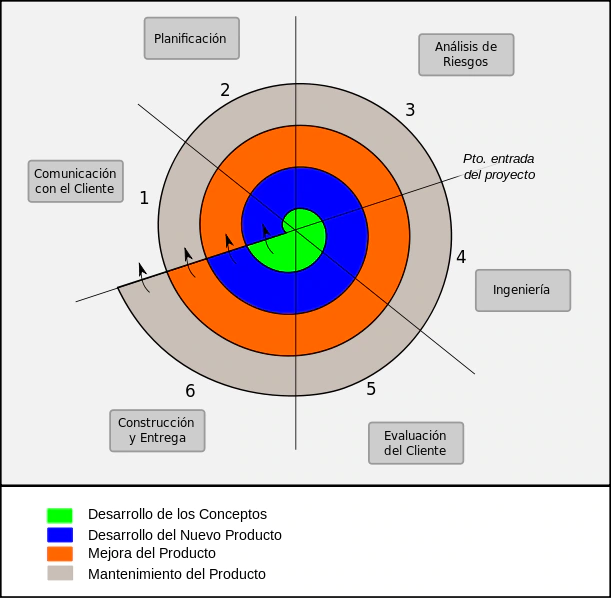
# **Planificación de las pruebas**

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 impuestas por el usuario.

En el proceso de desarrollo de software, nos vamos a encontrar con un conjunto de actividades, dónde 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. Con 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.

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.



## **1.1 Modelo en Espiral**

Se utiliza con éxito en proyectos donde el coste de un fallo es un gran riesgo, de ahí que su principal aportación sea considerar la gestión de esos riesgos, algo que en los modelos anteriores ni siquiera se menciona.

En concreto, los proyectos ejecutados con el modelo en espiral empiezan siendo pequeños, **investigando los mayores riesgos que se pueden tolerar, para pasar a agrandarse poco a poco**, en base a elementos clave sobre los que se construyen las siguientes fases de la espiral. Habitualmente tiene sentido aplicar este método en proyectos grandes, largos, caros y complejos.

En cuanto a su ejecución, **el modelo en espiral consiste en seguir ciclos crecientes de cuatro fases cada uno, que se van realizando siguiendo una forma de espiral**. En cada ciclo se pasa por dichas fases bien definidas, como en el modelo de cascada, pero con capacidad de evolucionar su complejidad con cada ciclo. Por tanto, se trata de un modelo evolutivo que, conforme avancen los ciclos, aumentará el tiempo de ejecución, así como el volumen de código fuente desarrollado y la complejidad de la gestión de riesgos y de la planificación.

### **1.1.1 Fases Modelo en Espiral**

1. **Planificación**. Se determinan los objetivos y el alcance del ciclo que comienza, tras un necesario ejercicio de investigación. Con cada iteración, se irá incrementando el tamaño de software entregado y la funcionalidad cubierta.
2. **Análisis de Riesgo.** Se evalúa todo aquello que pueda afectar al proyecto según el estado en que se encuentre y su grado de avance. Para ello, se diseñarán los prototipos que deberán ser validados en el ciclo.
3. **Implementación**. Se desarrolla y valida el software según el alcance acordado, el cual está íntimamente relacionado y condicionado con el análisis de riesgos anterior.
4. **Evaluación**. Antes de proceder a realizar otra vuelta en la espiral, se debe prestar atención a lo que sucedió en la vuelta anterior. Se debe analizar en detalle si los riesgos detectados anteriormente ya tuvieron solución. Básicamente, esta fase servirá para determinar el avance del proyecto y dar pistas de hacia dónde debe enfocarse la próxima iteración.

### **1.1.2 Ventajas del modelo en espiral**

1. Los factores de riesgo son reducidos.
2. El desarrollo es iterativo y se pueden incorporar funcionalidades progresivamente.

### **1.1.3 Desventajas del modelo en espiral**

1. La duración de la ejecución no es concreta.
2. Fallos en el análisis de riesgos podría influir negativamente a todo el proyecto.

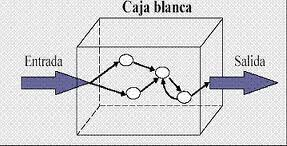
# **Tipos de pruebas**

No existe una clasificación oficial o formal, sobre los diversos tipos de pruebas de software. En la ingeniería del software, nos encontramos con dos enfoques fundamentales:

* **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. Estas pruebas pueden ser realizadas por personas ajenas al desarrollo del producto.



* **Prueba de la Caja Blanca (White Box Testing):** en este caso se prueba la aplicación desde dentro, usando su lógica de aplicación. Suelen ser realizadas por los desarrolladores o personal cualificado.



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.

En contraposición a lo anterior, 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.

**Reflexiona**

Resulta habitual, que en una empresa de desarrollo de software se gaste el 40 por ciento del esfuerzo de desarrollo en la prueba ¿Por qué es tan importante la prueba? ¿Qué tipos de errores se intentan solucionar con las pruebas?

## **2.1 Funcionales**

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

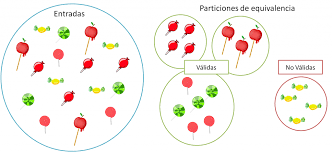
Las pruebas funcionales siguen el enfoque de las pruebas de Caja Negra. Comprenderían aquellas actividades cuyo objetivo sea verificar una acción específica o funcional dentro del código de una aplicación. Las pruebas funcionales intentarían responder a las preguntas ¿puede el usuario hacer esto? o ¿funciona esta utilidad de la aplicación?

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

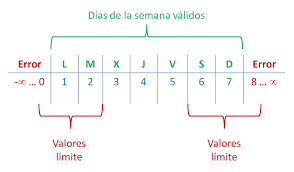
Si por ejemplo, estamos implementando una aplicación que realiza un determinado cálculo científico, en el enfoque de las pruebas funcionales, solo nos interesa verificar que ante una determinada entrada a ese programa el resultado de la ejecución del mismo devuelve como resultado los datos esperados.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

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

* **Particiones equivalentes:** la idea de este tipo de pruebas funcionales es considerar el menor número posible de casos de pruebas, para ello, cada caso de prueba tiene que abarcar el mayor número posible de entradas diferentes. Lo que se pretende, es crear un conjunto de clases de equivalencia, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase. Si tuvieramos dos entradas A y B, ambas con posibles valores positivos y negativos, probaríamos: +A +B , -A +B , +A –B , -A -B.



* **Análisis de valores límite:** en este caso, a la hora de implementar un caso de prueba, se van a elegir como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia.



* **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. Este tipo de pruebas, se suelen utilizar en aplicaciones que no sean interactivas, ya que es muy difícil generar las secuencias de entrada adecuadas de prueba, para entornos interactivos.
* **Pruebas de integración:** Las pruebas de integración son pruebas funcionales entre dos o más sistemas. El objetivo de las pruebas de integración es verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos.

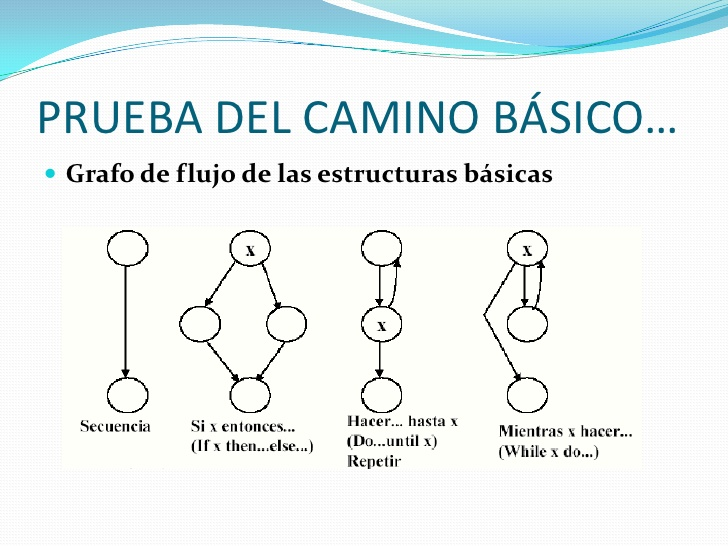
Existe otros tipos de pruebas funcionales, aunque todas comparten un mismo objetivo, y es comprobar, sólo actuando en la interfaz de la aplicación, que los resultados que produce son los correctos en función de las entradas que se le introducen para probarlos.

## **2.2 Estructurales**

Ya hemos visto que las pruebas funcionales se centran en resultados, en lo que la aplicación hace, pero no en cómo lo hace.

Para ver cómo el programa se va ejecutando, y así comprobar su corrección, se utilizan las pruebas estructurales, que se fijan en los caminos que se pueden recorrer:

Las pruebas estructurales son el conjunto de pruebas de la Caja Blanca. Con este tipo de pruebas, se pretende verificar la estructura interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo. Este tipo de pruebas, no pretenden comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas las instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer, etc.



Este tipo de pruebas, se basan en unos criterios de cobertura lógica, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores. Los criterios de cobertura que se siguen son:

* **Cobertura de sentencias**: se han de generar casos de pruebas suficientes para que cada instrucción del programa sea ejecutada, al menos, una vez.
* **Cobertura de decisiones:** se trata de crear los suficientes casos de prueba para que cada opción resultado de una prueba lógica del programa, 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.
* **Cobertura del camino de prueba:** se pueden realizar dos variantes, una indica que cada bucle se debe ejecutar sólo una vez, ya que hacerlo más veces no aumenta la efectividad de la prueba y otra que recomienda que se pruebe cada bucle tres veces: la primera sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces.

**Autoevaluación**

En las pruebas de caja negra:

* Es necesario conocer el código fuente del programa, para realizar las pruebas.
* Se comprueba que todos los caminos del programa se pueden recorrer, al menos una vez.
* Se comprueba que los resultados de una aplicación son los esperados para las entradas que se le han proporcionado.
* Es incompatible con la prueba de caja blanca.

## **2.3 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. El objetivo de las pruebas de regresión es comprobar que los cambios sobre un componente de una aplicación, no introduce un comportamiento no deseado o errores adicionales en otros componentes no modificados.

* Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error, como para realizar una mejora. No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.

Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se hayan realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

En un contexto más amplio, las pruebas de software con éxito, son aquellas que dan como resultado el descubrimiento de errores. Como consecuencia del descubrimiento de errores, se procede a su corrección, lo que implica la modificación de algún componente del software que se está desarrollando, tanto del programa, de la documentación y de los datos que lo soportan. La prueba de regresión es la que nos ayuda a asegurar que estos cambios no introducen un comportamiento no deseado o errores adicionales. La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas.

El conjunto de pruebas de regresión contiene tres clases diferentes de clases de prueba:

* Una muestra representativa de pruebas que ejercite todas las funciones del software;
* Pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio;
* Pruebas que se centran en los componentes del software que han cambiado.

Para evitar que el número de pruebas de regresión crezca demasiado, se deben de diseñar para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.

**Autoevaluación**

La prueba de regresión:

* Se realiza una vez finalizado cada módulo del sistema a desarrollar.
* Solo utiliza el enfoque de la caja negra.
* Se realiza cuando se produce una modificación, debido a la detección de algún error, en la fase de prueba.
* Es incompatible con la prueba de caja blanca.

# **Procedimientos y casos de prueba**

Según el IEEE, un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular, como, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito, incluyendo toda la documentación asociada.

Dada la complejidad de las aplicaciones informáticas, que se desarrollan en la actualidad, es prácticamente imposible, probar todas las combinaciones que se pueden dar dentro de un programa o entre un programa y las aplicaciones que pueden interactuar con él. Por este motivo, en el diseño de los casos de prueba, siempre es necesario asegurar que con ellos se obtiene un nivel aceptable de probabilidad de que se detectarán los errores existentes.

Las pruebas deben buscar un compromiso entre la cantidad de recursos que se consumirán en el proceso de prueba, y la probabilidad obtenida de que se detecten los errores existentes.

Existen varios procedimientos para el diseño de los casos de prueba:

* **Enfoque funcional o de caja negra**. En este tipo de prueba, nos centramos en que el programa, o parte del programa que estamos probando, recibe una entrada de forma adecuada y se produce una salida correcta, así como que la integridad de la información externa se mantiene. La prueba no verifica el proceso, solo los resultados.
* **Enfoque estructural o caja blanca**. En este tipo de pruebas, debemos centrar en la implementación interna del programa. Se trata de comprobar que la operación interna se ajusta a las especificaciones. En esta prueba, se deberían de probar todos los caminos que puede seguir la ejecución del programa.
* **Enfoque aleatorio.** A partir de modelos obtenidos estadísticamente, se elaboran casos de prueba que prueben las entradas del programa.

# **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, ayudándonos a realizar pruebas tanto estructurales como funcionales

Durante el proceso de desarrollo de software, se pueden producir dos tipos de errores: errores de compilación o errores lógicos. Cuando se desarrolla una aplicación en un IDE, ya sea Visual Studio, Eclipse o Netbeans, si al escribir una sentencia, olvidamos un ";", hacemos referencia a una variable inexistente o utilizamos una sentencia incorrecta, se produce un error de compilación. 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.

Para solucionar este tipo de problemas, los entornos de desarrollo incorporan una herramienta conocida como depurador. 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.

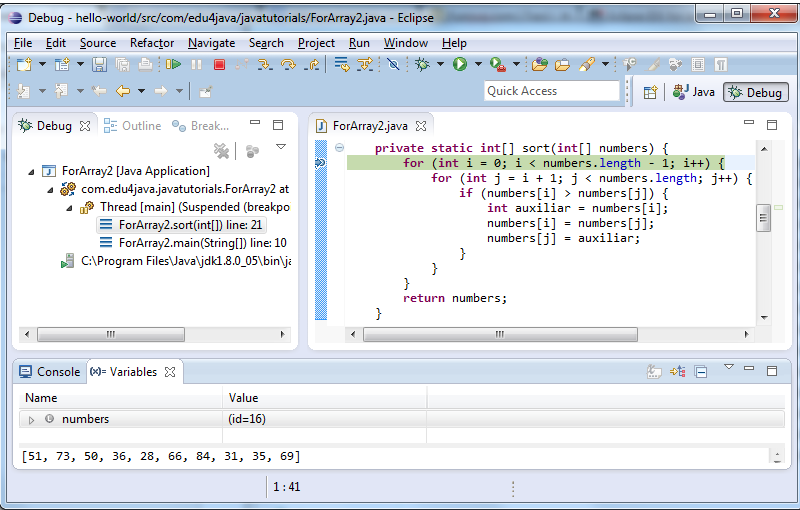
## **4.1. Puntos de ruptura**

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.



## **4.2. 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 los siguientes tipos 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 parámetros 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 (establecemos un punto de ruptura), 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.

Los distintos modos de ejecución se van a ajustar a las necesidades de depuración que tengamos en cada momento. Si hemos probada un método, y sabemos que funciona correctamente, no es necesario realizar una ejecución paso a paso en él.

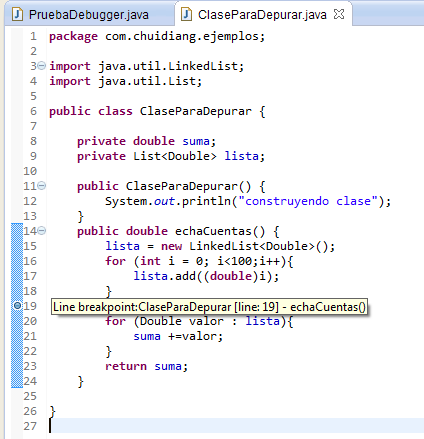
## **4.3. Examinadores de variables**

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.

## Poner un Breakpoint

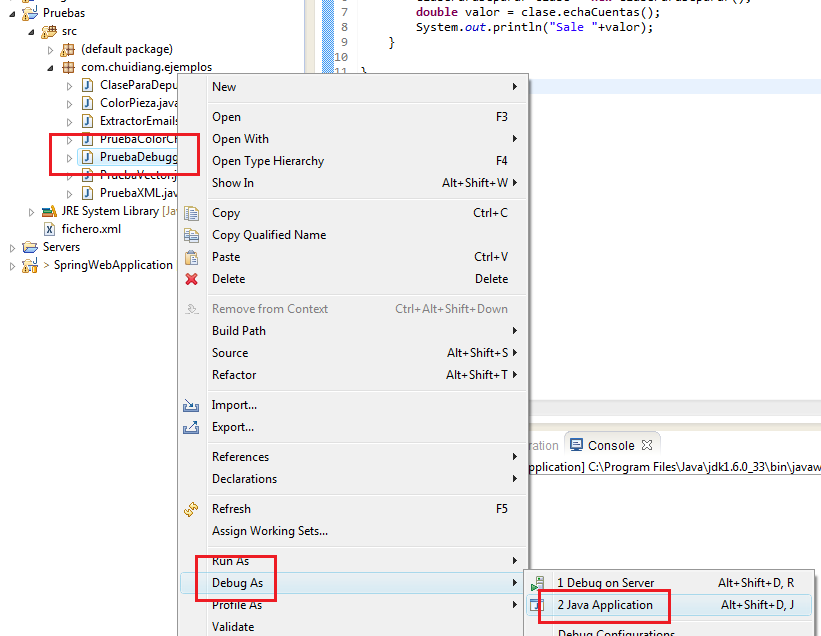
Un Breakpoint es un punto en nuestro código en el que al ejecutar con debugger, la ejecución se detendrá y podremos ver los valores que tienen las variables en ese momento, modificarlos, etc, etc. Poner un BreakPoint es sencillo, basta ir a nuestra clase y hacer doble click en el margen izquierdo del editor, justo en la línea donde queremos que la ejecución se detenga. En la figura vemos un circulito a la izquierda de la línea 19, ese es el breakpoink que acabamos de poner haciendo doble click justo ahí.



Si queremos quitarlo, nos basta con hacer doble click en él nuevamente.

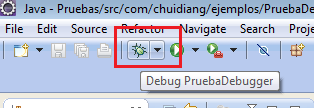
## Arrancar el programa en modo debugger

Para arrancar el programa en modo debugger, sacamos el menú con el botón derecho del ratón sobre la clase que tiene el main. Buscamos "debug as" -> "java application"



Esto arrancará nuestro programa en modo debugger y se ejecutará de forma normal hasta que la ejecución pase por un Breakpoint, en ese momento se detendrá.

Este proceso del menú y "debug as" -> "Java application" sólo es necesario la primera vez. Las siguientes veces que queramos ejecutar, podemos usar el botón en la parte superior con el dibujo de una cucaracha (un "bug")



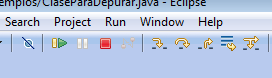
## La vista de debugger

Cuando arranca el programa en modo debug, eclipse posiblemente nos pregunte si queremos pasar a la vista de debug. Le indicamos que sí, marcando el check de "Remember my decision" si no queremos que nos lo pregunte cada vez que arranquemos el debugger sin estar en esa vista.

El programa se ejecuará de forma normal hasta llegar a un breakpoint. Es ese momento se parará y tendremos una serie de "cosas" útiles en la vista de debugger para poder ver variables, hacer que el programa se ejecute paso a paso, ver quién llama a quién, etc.

### **Botones para ir paso a paso**

En la vista de debugger tenemos botones para ir ejecutando paso a paso a partir del breakpoint donde nos hemos parado. Están en la parte superior y son como unas flechitas variadas amarillas.



El primer botón de la imagen, un breakpoint tachado, deshabilita temporalmente todos los breakpoints. Si hacemos que nuestro código corra, no se parará en ningún breakpoint.

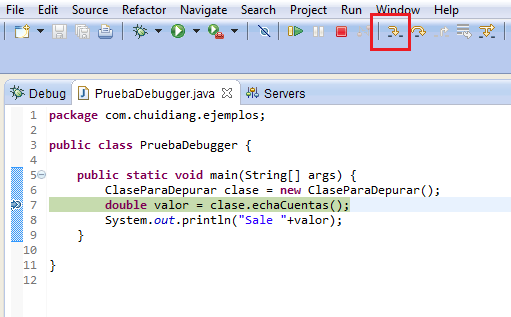
El siguiente botón, una flecha verde parecido al "play" de cualquier reproductor de música/video, hace que continue la ejecución de nuestro programa hasta el siguiente Breakpoint.

El siguiente botón, parecido al botón de pausa de cualquier reproductor de música/video, suspende la ejecución del programa que está corriendo. Es una forma de parar "al vuelo" el programa, esté donde esté en ese momento, haya o no breakpoint.

El cuadrado rojo es el botón de stop, aborta la ejecución de nuestro programa.

El siguiente botón que apenas se ve y cuyo tooltip es "disconnect" no tiene mucha utilidad en nuestro caso. Eclipse ofrece la posibilidad de conectarse a un programa java que se está ejecutando fuera de eclipse para depurarlo. Este botón nos desconectaría de ese programa remoto que estamos depurando.

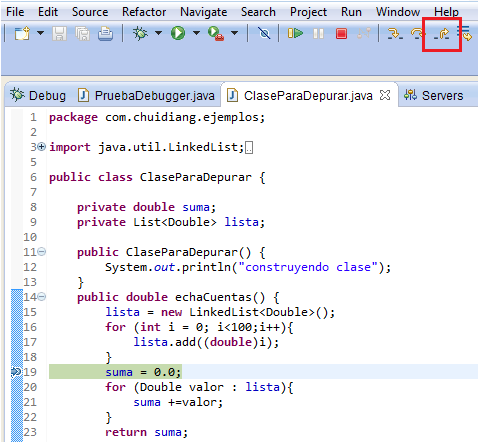
El siguiente botón, una flecha amarilla que se "mete" entre dos puntos negros, imagina que nuestro programa tiene el breakpoint en una línea en la que se hace una llamada a un método y se ha parado ahí. Pulsando este botón, el debugger se meterá dentro del método y se parará en la primera línea del mismo, de forma que podemos depurar dentro del método. Por ejemplo, si el breakpoint está donde se muestra en la siguiente figura.



al pulsar el botón marcado en rojo, el debugger se meterá dentro del método echaCuentas() y se parará en la primera línea de ese método.

El siguiente botón, una flecha amarilla que salta sobre un punto negro, avanza un paso la ejecución del programa, pero sin meterse dentro del método que encuentre. Por ejemplo, en la figura anterior, si pulsáramos este botón, la ejecución pasaría a la línea System.out.println(), sin pararse dentro del método echaCuentas().

Finalmente, el último botón es una flecha saliendo de entre dos puntos negros, hace avanzar el programa hasta que nos salimos del método actual y vamos hasta el sitio donde ha sido llamado. Por ejemplo, si el debugger está parado donde la figura y pulsamos este botón.



el debugger pasará justo al código donde se está llamando a este método, es decir, al echaCuentas() anterior.

# **Validaciones.**

En el proceso de validación interviene de manera decisiva el cliente. Hay que tener en cuenta, que estamos desarrollando una aplicación para terceros, y que son estos los que deciden si la aplicación se ajusta a los requerimientos establecidos en el análisis.

En la validación intentan descubrir errores, pero desde el punto de vista de los requisitos.

La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Un plan de prueba traza la clase de pruebas que se han de llevar a cabo, y un procedimiento de prueba define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos. Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que las documentaciones son correctas e inteligible y que se alcanzan otros requisitos, como portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento etc.

Cuando se procede con cada caso de prueba de validación, puede darse una de las dos condiciones siguientes:

* Las características de funcionamiento o rendimiento están de acuerdo con las especificaciones y son aceptables o Se descubre una desviación de las especificaciones y se crea una lista de deficiencias.
* Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de la terminación planificada.