

UNIDAD DIDÁCTICA 3

DISEÑO Y REALIZACIÓN DE PRUEBAS

**MÓDULO PROFESIONAL:
ENTORNOS DE DESARROLLO**



CESUR
Tu Centro Oficial de FP

Índice

RESUMEN INTRODUCTORIO	2
INTRODUCCIÓN	2
CASO INTRODUCTORIO	3
1. PROCEDIMIENTO DE PRUEBAS Y PLAN DE PRUEBAS	4
1.1 Estrategias de aplicación de pruebas	6
1.2 Tipos de pruebas funcionales, estructurales y regresión	7
1.3 Pruebas de caja blanca y caja negra	8
1.4 Planificación de pruebas	10
1.4.1 Pruebas de unidad.....	10
1.4.2 Pruebas de integración	11
1.4.3 Pruebas de validación	12
1.4.4 Pruebas de sistema	12
1.4.5 Pruebas de aceptación	12
1.5 Automatización de pruebas	13
1.6 Incidencias en los casos de prueba.....	14
1.7 Dobles de prueba. Tipos. Características.....	16
2. HERRAMIENTAS DE DEPURACIÓN	18
2.1 Consejos para la depuración.....	19
2.2 Análisis causal	26
3. MEDIDAS DE CALIDAD EN EL SOFTWARE	27
3.1 Normas de calidad	27
3.2 Software de evaluación de calidad del código. SonarLint	29
4. JUNIT. EJEMPLO	34
4.1 Crear proyecto en Eclipse	35
4.2 Crear clase Suma.....	38
RESUMEN FINAL	41

RESUMEN INTRODUCTORIO

A lo largo de la unidad conoceremos los conceptos relacionados con las pruebas de software y sus diferentes procedimientos.

Comenzaremos conociendo tanto el procedimiento como el plan de las pruebas a nivel de aplicación, las cuáles serán aplicadas según la funcionalidad a probar, conociendo también sus estrategias, tipos, planificación y su automatización.

Continuaremos estudiando las diferentes herramientas de depuración, conociendo sus consejos y el análisis causal.

También conoceremos las medidas de calidad en el software, las normas de calidad y, por último, ejemplos de uso del software JUnit y cómo crear un proyecto en Eclipse o una clase suma.

INTRODUCCIÓN

En el desarrollo de software una de las partes más importantes es la de probar el software que se está desarrollando. Para ello, es indispensable conocer el tipo de pruebas que podemos tener dentro de un proyecto, donde aplicarlas, cómo aplicarlas, etc.

Los planes de pruebas deben planificarse para que esta parte del proceso sea exitosa y el software llegue con la mejor calidad posible y con los menos errores posibles.

La automatización de pruebas en los grandes proyectos puede llevarnos a reducir el tiempo y el dinero en el desarrollo de software, por lo que se deben conocer herramientas que permitan repetir automáticamente las pruebas con cada cambio de envergadura que se lleve dentro del proyecto.

CASO INTRODUCTORIO

El equipo de desarrollo en el que trabajas ha generado la primera versión de una aplicación para gestionar varias tiendas de ropa. A priori, toda la funcionalidad ha sido probada por los propios programadores a medida que han ido implementando los requisitos. Sin embargo, se hace necesario que el equipo de pruebas establezca un plan de pruebas, que consiste en analizar la aplicación y crear tantos casos de prueba como sean necesarios. Por ejemplo, para loguearse en el aplicativo, es necesario probar con un usuario y una contraseña válida y comprobar si el resultado esperado es el mismo que el resultado obtenido. En caso afirmativo, la prueba verifica el funcionamiento del login. Pero también se hace necesario probar con usuarios y contraseñas no válidas, jugando con las diversas casuísticas (usuario correcto, contraseña no válida; usuario incorrecto, contraseña no válida, etc.).

Al finalizar la unidad serás capaz de realizar las pruebas necesarias para comprobar que la aplicación cumple con los requisitos del cliente, para ello planificarás el plan de pruebas del software, sabiendo las normas de calidad del software vigentes y la relación entre las pruebas y la calidad del software.

1. PROCEDIMIENTO DE PRUEBAS Y PLAN DE PRUEBAS

Para poder llevar a cabo la fase de pruebas de la aplicación de gestión de tiendas de ropa, es importante saber el tipo de pruebas que vas a realizar sobre tu software, por lo que valoras qué tipo de prueba se adecua más a lo que necesitas actualmente.

Además, sabes que la automatización de pruebas es un paso importante que te puede ahorrar mucho tiempo y dinero en el desarrollo de software, es por ello que debes conocer las posibilidades y componentes necesarios que vas a utilizar en este tipo de automatización.

Un procedimiento de prueba es la definición de lo que se desea conseguir con las pruebas, qué es lo que va a probarse y cómo.

El objetivo principal de las pruebas no es siempre detectar errores, muchas veces lo que quiere conseguirse es un rendimiento determinado, que el programa tenga cierta apariencia, o cumpla ciertas características...

Por lo que, si nuestro programa no tiene errores en las pruebas, no significa que el software supere el procedimiento, ya que hay muchos parámetros en juego.

Al diseñar los procedimientos, se eligen los parámetros en los que van a realizarse las pruebas, y las personas que van a hacerlas. En este sentido, no serán siempre los programadores los que realicen las pruebas, siempre debe haber personal externo al equipo de desarrollo, ya que los programadores solo prueban las cosas que funcionan, si supieran donde están los errores los corregirían.

Debemos tener en cuenta que es muy difícil, por no decir imposible, probar todo y muchos de los errores del software que se está desarrollando, saldrán cuando el software ya esté implantado.

Se denomina **plan de pruebas** a un conjunto de acciones y pruebas que cumplen con las siguientes características:

- Debe especificar de una forma correcta el **objetivo** de la prueba, por ejemplo, comprobar propiedades del software como corrección, robustez, fiabilidad, amigabilidad, etc.
- Especificar la **medida** en la que se mostrará el resultado.

- Además del objetivo, se debe dejar claro **en qué va a consistir la prueba** hasta el último detalle, desde las entradas que pueda tener la prueba, su ejecución, las salidas, etc.
- En el plan de pruebas y más concretamente en cada caso de prueba, se debe definir siempre cuál es el **resultado esperado**.



PARA SABER MÁS

Aprende más sobre el plan de pruebas y cada uno de los tipos de pruebas:



Diferentes pruebas que se deben incluir en el plan de pruebas:

- Se deben realizar pruebas donde los valores de entrada y salida correspondan con **valores límite**, para analizar condiciones límite que se puedan producir.
- Pruebas con **valores normales**, diferentes de los límites, pero utilizando un amplio rango de valores.
- Realizar **pruebas de error**, es decir, pruebas basadas en datos con los que se presupone que se producirán errores.
- Normalmente se deben añadir **pruebas especiales** que dependerán del objetivo que se ha definido en el plan de pruebas. Se deberán, por tanto, diferenciar pruebas para comprobar la robustez, la velocidad a la que se ejecuta la aplicación, etc.

1.1 Estrategias de aplicación de pruebas

Las pruebas comienzan a nivel de módulo. Una vez terminadas, progresan hacia la integración del sistema completo y su instalación. Culminan cuando el cliente acepta el producto y se continua a su explotación inmediata.

Los objetivos del documento del plan de pruebas comprenden: señalar el enfoque, los recursos y el esquema de actividad desde prueba, así como los elementos a probar, las características, las actividades de prueba, el personal responsable y los riesgos asociados.

El esquema del documento del plan de pruebas es el siguiente:

1. Identificador único del documento.
2. Introducción y resumen de elementos y características a probar.
3. Elementos software a probar.
4. Características a probar.
5. Características que no se probarán (si fuera necesario).
6. Enfoque general de la prueba.
7. Criterios de paso/fallo para cada elemento.
8. Criterios de suspensión y requisitos de reanudación.
9. Documentos a entregar.
10. Actividades de preparación y ejecución de pruebas.
11. Necesidades de entorno.
12. Responsabilidades en la organización y realización de las pruebas.
13. Necesidades de personal y formación.
14. Esquema de tiempos.
15. Riesgos asumidos por el plan y planes de contingencias.
16. Aprobaciones y firmas con nombre y puesto desempeñado.

El objetivo del documento de especificación del diseño de pruebas especificar los procesos necesarios e identificar las características que se deben probar con el diseño de pruebas.

El objetivo del documento de especificación de caso de prueba trata de definir al menos uno de los casos de prueba identificando una especificación del diseño de las pruebas.

El objetivo del documento de especificación de procedimientos de prueba pretende especificar los distintos pasos para la ejecución de, al menos, un conjunto de casos de prueba o los pasos utilizados para analizar un elemento software con el propósito de evaluar un conjunto de características del mismo.

1.2 Tipos de pruebas funcionales, estructurales y regresión

En cuanto al tipo de pruebas por realizar, existen muchas categorías, vamos a ver las más frecuentes:

- **Pruebas funcionales:** buscan que los componentes software diseñados cumplan con la función con la que fueron diseñados y desarrollados. Estas pruebas buscan lo que el sistema hace, más que cómo lo hace.

La persona que realiza las pruebas (tester o ingeniero de pruebas), se basa en la documentación existente (manual de usuario y otros manuales) junto con los usuarios, ya que son los que saben cómo tiene que funcionar el software.

Estas pruebas suelen considerarse como pruebas de caja negra. No se evalúa cómo el sistema funciona internamente, pero sí qué es lo que hace.

- **Pruebas no funcionales:** son aquellas pruebas más técnicas que se realizan al sistema, como por ejemplo pruebas de carga, pruebas de estrés, pruebas de rendimiento, pruebas de fiabilidad, etc. Estas siguen siendo de caja negra, puesto que nunca se examina la lógica interna de la aplicación.
- **Pruebas estructurales:** son pruebas que examinan de forma más detallada la arquitectura de la aplicación, son de caja blanca, ya que, en algún momento, se utilizan técnicas de análisis del código. Generalmente, para este tipo de pruebas, se utilizan herramientas especializadas.
- **Pruebas de regresión o pruebas repetidas:** repetición de pruebas ya realizadas una vez que se ha desarrollado una modificación en el software. Estas pruebas intentan descubrir si existe algún error tras las modificaciones o si se encuentra algún tipo de problema que no se había descubierto previamente.

En la mayoría de los casos, este tipo de pruebas no sirve para detectar errores, sino para ver que, tras la modificación de código, hay ausencia de los mismos.

Suelen ser pruebas que se automatizan y se agrupan en conjuntos llamados conjuntos de pruebas de regresión (regression test suites).

1.3 Pruebas de caja blanca y caja negra

Ahora comprobaremos cuáles son las pruebas denominadas de “caja blanca” y de “caja negra”:

1. PRUEBAS CAJA BLANCA.

En las pruebas de caja blanca (clear box testing) conocemos o tenemos en cuenta el código a probar. En este tipo de pruebas, la persona que realiza las pruebas está en contacto con el código fuente y tiene el objetivo de probar el código y cada uno de sus elementos.

Existen algunas clases de pruebas de este tipo como, por ejemplo:

- **Pruebas de cubrimiento:** consisten en ejecutar todas las líneas de código de las que este formado el programa, aunque hay algunos aspectos no se prueben nunca, ya que las condiciones para acceder a esa parte del código no se lleguen a dar, algún condicional o excepción.

Para realizar las pruebas, habrá que generar los suficientes casos de prueba para poder cubrir los distintos caminos independientes del código.

- **Pruebas de condiciones:** en una condición, puede haber varias condiciones y se tendrán que crear distintos casos de pruebas para cada condición (operador lógico o de comparación), que se tengan.
- **Pruebas de bucles:** se basará en la repetición de un número especial de veces. A la hora de probar los bucles simples, hay que tener en cuenta:
 - Repetir el número de repeticiones máximo, máximo -1 y máximo +1 veces el bucle.
 - Repetir el bucle 0 y bucle 1.
 - Repetir el bucle un número indeterminado de veces.

2. PRUEBAS CAJA NEGRA.

Las pruebas de caja negra son aquellas que simplemente prueban la interfaz sin tener en cuenta el código, dentro de estas pruebas podemos encontrarnos con:

- **Pruebas de clases de equivalencia de datos.**

Un ejemplo claro de este tipo de pruebas es la prueba de una interfaz con un usuario y una contraseña. El usuario debe tener mayúsculas y minúsculas, no puede tener caracteres que no sean alfabéticos y ha de tener, entre 6 y 12 caracteres. Las contraseñas tendrán, entre 8 – 10 caracteres, contendrán letras y números.

Para ello se deberán crear casos de prueba con valores para el usuario y para la contraseña que cumplan las características de los mismos y valores que no los cumplan.

- **Pruebas de valores límite.**

El objetivo es generar valores que puedan probar si la interfaz y el programa funcionan correctamente en valores límite, sobrepasándolos, no sobrepasándolos, y estando justo en el límite.

Vamos a imaginar una web de un banco en la que podemos hacer transferencias, queremos probar una interfaz que nos aparece si sobrepasamos el límite de las transferencias fijado en 10.000€, para ello generaríamos casos de prueba en los que se intentarían hacer transferencias de, por ejemplo, valores fuera de rango -100 o 20.000, valores límite 0, 1, 9.999, 10.001, o valores intermedios como 9.000 o 2.500.

- **Pruebas de cubrimiento o interfaces:**

Este tipo de pruebas trata de cubrir todos los aspectos funcionales que debe desarrollar nuestra aplicación, es decir, se trata de testear la interfaz y los objetos que la forman introduciendo valores (a poder ser reales y no inventados) y probar las distintas funcionalidades que esta interfaz ofrece cubriendo todas las posibilidades de dicha interfaz.

Algunas de las pruebas que se pueden realizar en este sentido son:

- A. Seguir el manual del usuario:** el tester seguirá el manual del usuario para probar la aplicación, en este caso si la pasa, podrán hacerse casos de prueba más complicados.

- B. **Testear la usabilidad:** evalúa si el resultado de la interfaz es el esperado por el usuario, comprobar el feedback del usuario para saber si es una interfaz sencilla o difícil de controlar.

En este tipo de pruebas, es recomendable escuchar al usuario que va a usar la aplicación, ya que debemos hacerla lo más amigable posible, no solo para que le facilite el trabajo, sino para que la nueva aplicación no sufra rechazo.

- C. **Testear la accesibilidad:** al testear la accesibilidad, no solamente estamos probando que el software sea accesible a los usuarios con discapacidad, sino que también sea accesible por frameworks de test automatizados.

Se define un software accesible cuando el programa o aplicación se adecua a los usuarios con discapacidad, y pueden hacer su trabajo de forma efectiva, siendo la satisfacción con el software buena.

1.4 Planificación de pruebas

La planificación de pruebas es un punto importante a la hora de planificar un proyecto, siendo la toma de decisiones:

- ¿Qué tipo de pruebas se van a realizar?
- ¿Cuándo se van a realizar las pruebas?
- ¿Desde cuándo han de tenerse en cuenta las pruebas?

Si el proyecto en el que se está trabajando es grande, se tendrán en cuenta las pruebas que se van a enumerar a continuación, en el caso de que sean proyectos pequeños, será el responsable de proyecto el encargado de decidir qué tipo de pruebas se realizarán al proyecto, atendiendo al tiempo y presupuesto que se tenga sobre el mismo.

1.4.1 Pruebas de unidad

Son pruebas formales que permiten declarar que un módulo está listo y terminado. Se refiere una unidad de prueba a uno o más módulos que cumplen que:

- Todos son del mismo programa.
- Al menos uno de ellos no ha sido probado.
- El conjunto de módulos es el objeto de un proceso de prueba.
- Este tipo de pruebas se automatizan a través de herramientas como JUnit.

1.4.2 Pruebas de integración

Los módulos individualmente probados se integran para comprobar sus interfaces (comunicación con otros módulos) en el trabajo conjunto. Se han de tener en cuenta las interfaces entre componentes de la arquitectura del software.

Implican una progresión ordenada de pruebas que van desde los componentes o módulos y que culminan en el sistema completo.



ENLACE DE INTERÉS

Conoce un ejemplo de prueba real de software:



Tipos fundamentales de integración:

- **Integración incremental:** Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados.
- **Ascendente:** Se comienza por los módulos hoja y se va subiendo.
- **Descendente:** Se comienza por el módulo raíz y se va bajando.
- **Integración no incremental:** Se prueba cada módulo por separado y seguidamente se integran todos de una vez, probando todo el programa.

Habitualmente, las pruebas de unidad y de integración se realizan en el mismo tiempo. Al añadir un nuevo módulo, el software cambia y se establecen nuevos caminos de flujo de datos, nueva E/S y nueva lógica de control. Puede haber problemas con acciones que anteriormente funcionaban bien, por lo que se deberán ejecutar de nuevo el conjunto de pruebas que se han realizado anteriormente para asegurarse que los cambios no han dado lugar a cambios colaterales, estas pruebas son las llamadas **pruebas de regresión**.

1.4.3 Pruebas de validación

Se llevan a cabo cuando se han terminado las pruebas de integración. El software terminado se prueba como un conjunto para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimiento, seguridad, etc. La validación se consigue cuando el software funciona según las expectativas del cliente en función de los requisitos definidos.



PARA SABER MÁS

Comprueba la información brindada en un seminario de Verificación, Validación y Pruebas unitarias:



1.4.4 Pruebas de sistema

El software ya cumple con todos los requisitos, por tanto, ahora se debe integrar con el resto del sistema, instalando la aplicación en el mismo sistema operativo en el que el cliente lo va a utilizar y con las mismas características hardware, probando su funcionamiento. Se debe comprobar, además, la seguridad en el acceso a los datos, el rendimiento del software con grandes cargas de trabajo, la tolerancia a fallos en la red, etc. Se debe probar la recuperación del sistema en caso de errores en la aplicación que puedan hacer que finalice de forma no esperada, recoger los logs de la aplicación, etc.

Estas pruebas se deben ejecutar y observar su rendimiento en condiciones límite y de carga máxima (pruebas de rendimiento). Estas pruebas sirven para verificar que se han integrado adecuadamente todos los elementos del sistema y que se realizan las funciones apropiadas, realizándose sobre lo que se denomina entorno de preproducción.

1.4.5 Pruebas de aceptación

Se instala el software en el propio entorno de explotación (entorno de producción) y el cliente comprueba que el software cumple con los requisitos establecidos.

1.5 Automatización de pruebas

Para la automatización de pruebas se van a tener en cuenta 3 condicionantes: casos de prueba, procedimientos de prueba y componentes de prueba:

- **Casos de prueba:** cada caso de prueba debe definir el resultado de salida esperado que se comparará con el realmente obtenido.
- **Procedimientos de prueba:** especificar cómo y cuándo se realizarán las pruebas.
- **Componentes de prueba:** automatiza uno o varios procedimientos de prueba o parte de ellos.

Para describir los componentes de la prueba, sería de la siguiente forma:

- Por cada caso de prueba, implementar el código correspondiente en el componente.
- Se escribe el código del componente de tal manera que recorra en una Base de Datos los casos de prueba y los ejecute.
- Cada vez que se añada un caso de prueba, simplemente se añade en la Base de Datos, pero el código del componente no cambiaría.
- Se pueden usar entornos de trabajo disponibles para pruebas (JUnit).



ENLACE DE INTERÉS

Aunque más adelante en la unidad se estudiará una de las herramientas utilizadas para la automatización de pruebas (JUnit), puedes comprobar otra herramienta que también está disponible en el mercado, entre muchas otras:



1.6 Incidencias en los casos de prueba

Documentar incidencias de manera efectiva es esencial en el proceso de pruebas de desarrollo de aplicaciones ya que, a la hora de realizar pruebas, no siempre van a salir bien. A continuación, se pasa a detallar una guía paso a paso sobre cómo documentar estas incidencias:

1. **Herramienta de Seguimiento de Problemas:** utiliza una herramienta de seguimiento de problemas (bug tracking) como Jira, Bugzilla, Trello, o cualquier otra que se adapte a tus necesidades. De esta forma se llevará un control general de las incidencias donde los usuarios de estas herramientas podrán encontrar toda la información sobre el seguimiento y resolución de las mismas.
2. **Información General:** la incidencia debe presentar una información específica para saber tratarla:
 - a. **Título:** debe ser claro y conciso, lo que nos ayudará a identificar el problema de forma única.
 - b. **ID de Prueba:** número identificador asociado la incidencia con la prueba específica.
 - c. **Entorno:** Especificamos la plataforma en la que nos ha dado el problema (iOS, Android, Web, Windows, Linux, ...)
 - d. Indica la versión del **sistema operativo**.
 - e. **Fecha y Hora:** indicamos la fecha y hora de la incidencia.
3. **Descripción detallada:**
 - a. La **descripción** que se da del problema debe detallarse paso a paso, incluyendo detalles sobre las acciones que se realizan, ya que, muchas veces, cuando se están llevando a cabo casos de prueba, la persona que debe solucionar los problemas no es la misma que lo ha detectado. Al incluir detalles de la detección de problema, la persona que debe solucionarlo, podrá recrearlo para ponerle solución.
 - b. **Resultado Esperado:** se debe especificar qué resultado se esperaba de la prueba para corroborar que no se ha llegado al mismo y que la persona que lo corrige sepa qué resultado se esperaba para el caso en concreto.
 - c. **Resultado Observado:** describe lo que realmente sucedió y dónde se deben incluir mensajes de error si los hay.
 - d. **Capturas de Pantalla/Videos:** adjunta capturas de pantalla o grabaciones de video si es posible.

4. **Prioridad y Severidad:** establece la urgencia de la corrección (alta, media, baja) evaluando el impacto del problema en la funcionalidad general (crítico, mayor, menor).
5. **Categorización:** clasifica el problema según los parámetros que se establezcan en el proyecto (funcionalidad, usabilidad, rendimiento, etc.). Además, indica si la incidencia proviene de pruebas manuales o automatizadas.
6. **Información Adicional:**
 - a. **Dispositivo/Emulador:** especifica el dispositivo físico o emulador utilizado.
 - b. **Navegador (si aplica):** Indica el navegador y su versión.
 - c. **Datos del Usuario (si aplica):** proporciona detalles sobre la cuenta de usuario, permisos que tienes, etc.(si es necesario para reproducir el error).
7. **Reproducibilidad:** indica si el problema es reproducible y, en caso afirmativo, proporciona pasos detallados para reproducirlo.
8. **Asignación y Estado:** una vez que se crea la incidencia, se asigna a una persona o equipo que la resolverá o tramitará durante un periodo en el que se informará de la evolución de la misma:
 - a. **Asigna la Incidencia:** asegúrate de que la incidencia esté asignada a la persona adecuada.
 - b. **Estado:** actualiza el estado de la incidencia (abierto, en progreso, cerrado).
9. **Comunicación:** durante todo el proceso, se debe comentar lo que se va haciendo y tener notificaciones de los cambios de las incidencias.
10. **Verificación y Cierre:** una vez resuelta la incidencia, se debe verificar la solución para poder cerrar la misma. Para ello se suelen crear una serie de criterios de cierre.
11. **Historial:** deben quedar registrado todos los pasos que se han seguido para resolver la incidencia como tal en un historial.

1.7 Dobles de prueba. Tipos. Características.

Cuando hablamos de "dobles de prueba" en el contexto de la creación de software, nos referimos a entidades o componentes que se utilizan en lugar de otros elementos del sistema durante el proceso de prueba. Estos "dobles" se emplean para facilitar y mejorar la calidad del proceso de prueba. Se pasan a detallar algunos de los dobles de prueba que más se utilizan, junto a sus características:

- **Dummy Objects (Objetos ficticios):** son objetos de relleno que se utilizan como argumentos, pero no realizan ninguna operación significativa. Su objetivo es cumplir con la firma de la interfaz o clase a la que se pasan.
- **Fake Objects (Objetos falsos):** implementan una funcionalidad simplificada y pueden ser utilizados en pruebas de rendimiento o en escenarios donde no es práctico usar el sistema real. No son adecuados para entornos de producción, pero son más realistas que los objetos ficticios.
- **Stubs (Punteros):** Proporcionan respuestas predefinidas a las llamadas de métodos durante las pruebas. Se utilizan para simular partes específicas del sistema y verificar la interacción correcta con estas partes.
- **Mocks (Simulaciones):** Similar a los stubs, pero también verifican que las interacciones esperadas ocurran durante las pruebas. Se configuran con expectativas sobre cómo se utilizarán y responden en consecuencia.
- **Spies (Espías):** registran información sobre llamadas a métodos o funciones durante las pruebas. Permiten verificar que los métodos fueron llamados con los argumentos correctos y cuántas veces fueron invocados.
- **Dummies de base de datos:** se utilizan para simular el acceso a la base de datos durante las pruebas. Evitan la dependencia de la base de datos real, permitiendo pruebas más rápidas y aisladas.
- **Dobles temporales:** representan versiones simplificadas de componentes del sistema, especialmente útiles en entornos donde esos componentes no están disponibles.
- **Sandboxes (Entornos de prueba):** crean un entorno de prueba aislado que simula el entorno de producción tanto como sea posible. Pueden incluir bases de datos de prueba, servicios simulados, etc.

Las características específicas de los dobles de prueba pueden variar según el contexto y las herramientas utilizadas. La elección del tipo de doble dependerá de los requisitos de la prueba y del objetivo de asegurar un comportamiento correcto y confiable del software en desarrollo.



VÍDEO DE INTERÉS

Amplía la información sobre las ventajas de la automatización:



EJEMPLO PRÁCTICO

A nuestro equipo de trabajo llega una aplicación software que debe ser probada con cierta urgencia, ya que el cliente espera que le sea entregada y, por consecuencia, probada, en los próximos días. ¿Cómo procederías a realizar las diferentes pruebas?

Es imprescindible hacer uso del plan de pruebas implementado para dicha aplicación. En él se encontrarán los casos de prueba que se hayan definido. Cada caso de prueba tendrá un enunciado de la prueba, datos de la ejecución de la prueba (nombre del tester, fecha, hora, lugar, equipo en el que ha sido privado, etc.) un caso esperado y una zona para indicar el caso obtenido. Siempre que el caso esperado sea idéntico al caso obtenido se podrá decir que la prueba no ha conseguido encontrar ningún tipo de defecto. En muchas ocasiones, esta tarea se puede realizar de forma automática, mediante el uso y la configuración de herramientas como ALM de HP, entre otras.

2. HERRAMIENTAS DE DEPURACIÓN

Como profesional sabes que, a la hora de poder realizar pruebas, es importante tener herramientas que te permitan identificar por qué se ha producido un error. Por ello, para el desarrollo de vuestra aplicación de gestión de tiendas de ropa, has elegido Eclipse. Te dispones a formar a los integrantes del equipo en las herramientas que Eclipse proporciona para la depuración, dado que son herramientas que tienen todos los entornos de desarrollo.

Se conoce **depuración** al proceso de identificar y corregir defectos que pueda tener el software creado. Son muchos y muy frecuentes los errores humanos que se cometen mientras se escribe un código, aumentando bastante la complejidad de un software. Es por ello que la depuración se convierte en un aliado para los programadores bastante eficiente a la hora de corregir este tipo de problemas. Los compiladores tienen la posibilidad de ejecutar un programa paso por paso, permitiendo al desarrollador comprobar los valores intermedios de las variables, las posibles entradas, salidas, etc. Los depuradores se deben usar para realizar un seguimiento sobre el comportamiento dinámico de los programas.



¿SABÍAS QUE...?

Utiliza las diversas herramientas de depuración como ayuda para depurar aplicaciones en un entorno de desarrollo autónomo o colaborativo.

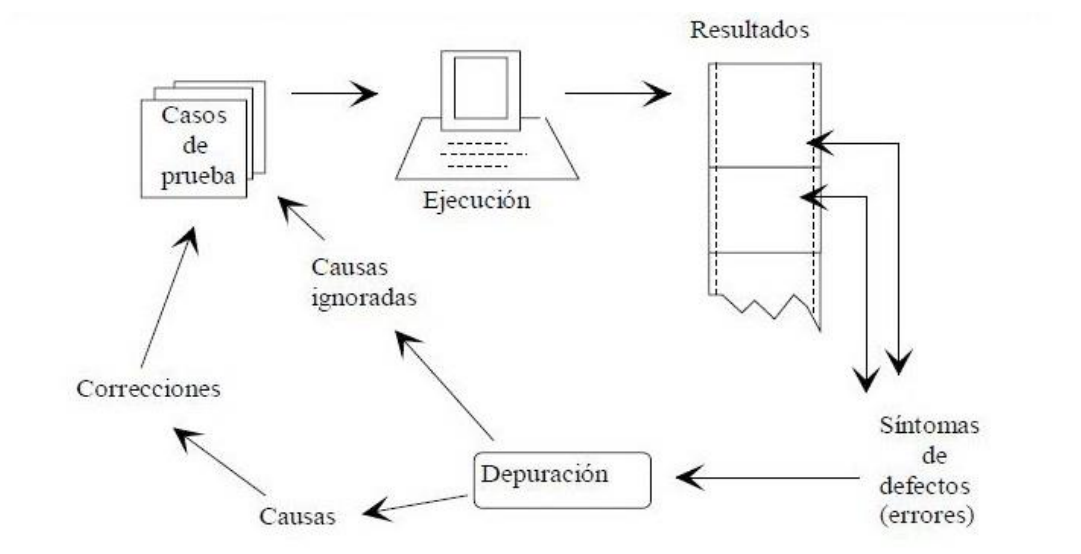
En la práctica, no obstante, su uso es tedioso y sólo serán eficaces si se persigue un objetivo claro y previamente definido o al menos previsible. Por norma general, su utilización es consecuencia de la detección de un error. Si el programa se comporta mal en un cierto punto, se debe averiguar la causa para poder repararlo. Esta causa a veces es inmediata, por ejemplo, una asignación de variables errónea o un operador equivocado.

Sin embargo, el error también puede depender del valor concreto de los datos en un cierto punto y hay que buscar el motivo. Antes de acudir al depurador, hay que delimitar lo máximo posible la zona donde se puede encontrar el fallo, identificar el dominio del mismo e investigar las propiedades de los datos que lo provocan.

2.1 Consejos para la depuración

Los consejos para realizar correctamente la depuración son:

- Analizar la información e intentar resolver los errores antes de depurar.
- Usar herramientas de depuración sólo como último recurso secundario.
- Se deben depurar los errores individualmente.
- Fijar la atención en los datos.
- Al corregir un error, volver a realizar todas las pruebas y volver a depurar.
- Al corregir un error puede que aparezcan otros.




Relación entre pruebas y depuración.

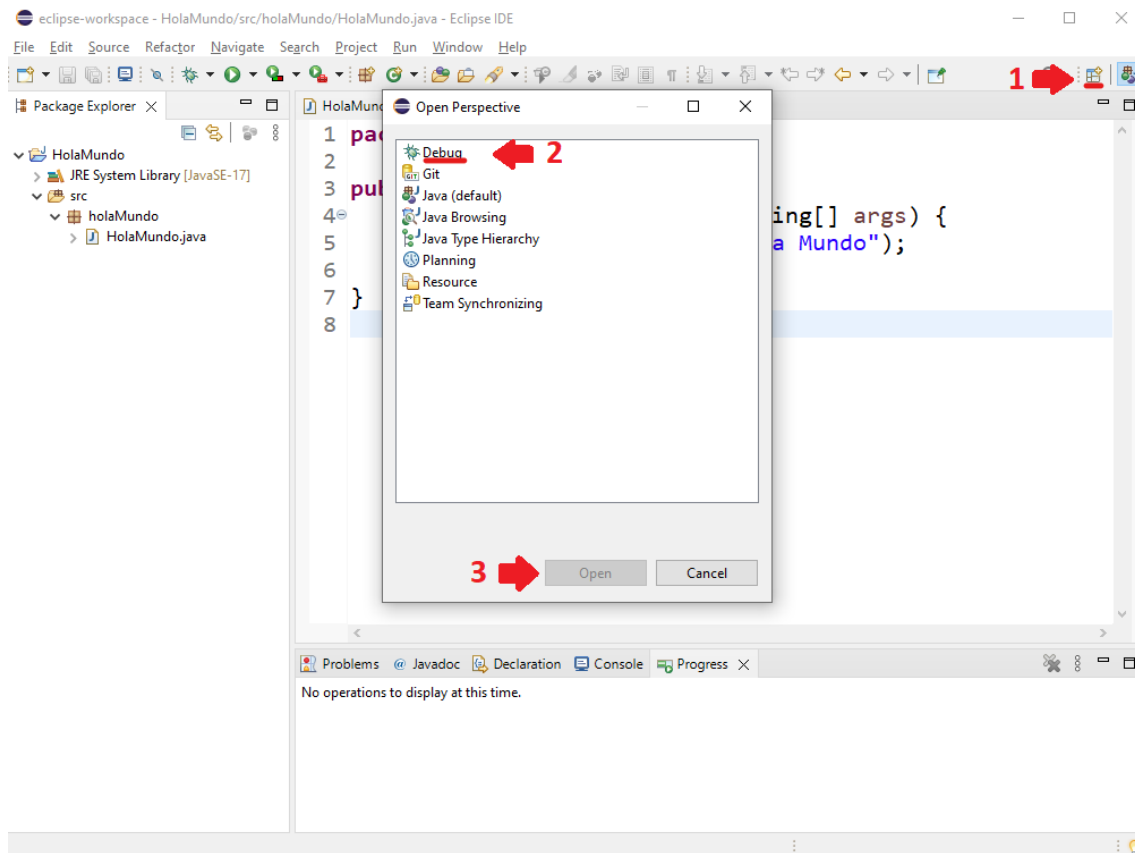
Ahora conoceremos cómo se lleva a cabo un proceso de depuración en **Eclipse**:

Un programa debe compilarse con éxito para poder utilizarlo en el depurador, es decir, en Eclipse no puede aparecer ningún error en rojo. El depurador nos permite 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. Para ello hay que conocer una serie de conocimientos para poder trabajar con Eclipse:

- **Vista Debugger de Eclipse.**

Cuando se va a depurar, hay que abrir una vista Debugger dentro del programa, hay dos formas de hacerlo. La primera presionando el botón , donde aparece un mensaje de advertencia diciendo que se va a cambiar de vista a una vista de tipo Debugger. La segunda presionando en la esquina superior izquierda de la vista principal de Eclipse (1).

Se abre una ventana y elegimos la opción Debug (2), por último, presionamos en “Open” (3).

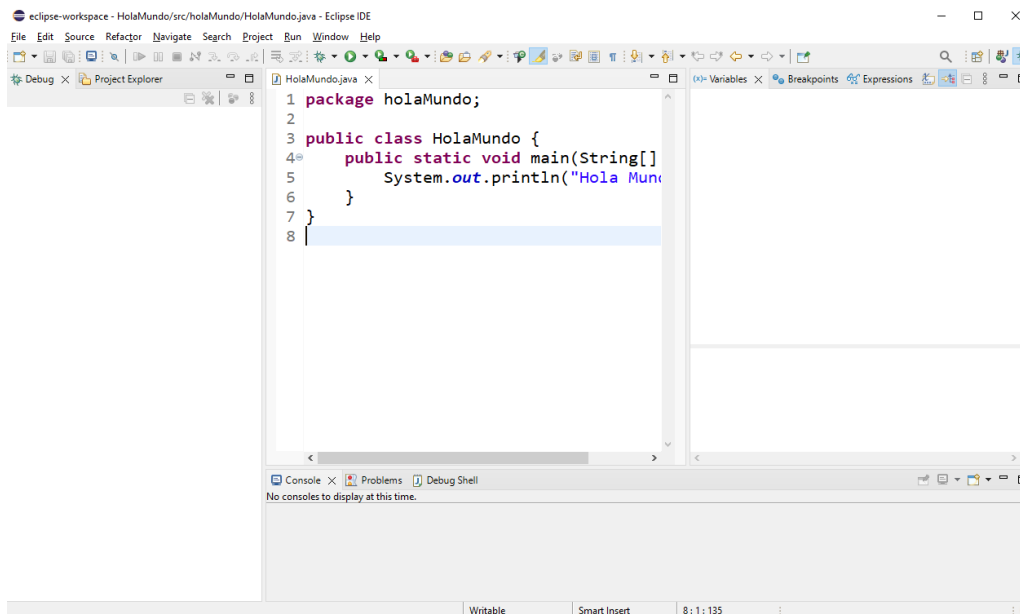


Abriendo perspectiva Debugger en Eclipse por primera vez.

Se nos abre una perspectiva nueva, en la que se puede apreciar de izquierda a derecha:

- **Pestaña Debug**, donde aparecen los procesos que se están ejecutando en Eclipse y los que se han ejecutado dentro de la sesión en la que estamos. Entendiendo como sesión el momento en que abrimos Eclipse y nos ponemos a ejecutar y probar.
- **Pestañas de archivos .java**, es el código que queremos probar, aquí hay que añadir los breakpoints o puntos de ruptura que queremos poner dentro del programa. Estos puntos de ruptura son momentos del programa en los que queremos parar la ejecución para ver cómo se comportan los valores de variables o expresiones que hay dentro del código.
- **Pestaña de Variables**, aquí podemos ver el valor de las variables que hay declaradas dentro del momento en el que se para el código (utilizando los breakpoints).

- **Pestaña Breakpoints**, tiene declarados todos los puntos de ruptura que hay en todos los archivos .java que se encuentran en el Workspace.
- **Expresiones**, nos muestra el valor de ciertas expresiones que hay en el código o que se añaden posteriormente.



Perspectiva Debugger Eclipse.

Una vez que se ha abierto esta perspectiva por primera vez, si queremos seguir desarrollando y cambiar a una perspectiva de programación más cómoda, volvemos al panel superior derecha y presionamos sobre el botón

Para volver a la vista debugger volvemos al panel superior derecho y presionamos sobre el botón

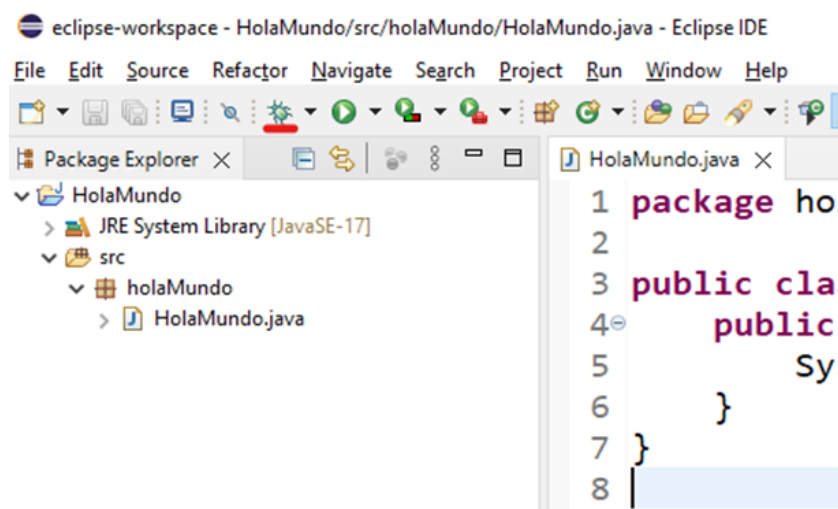
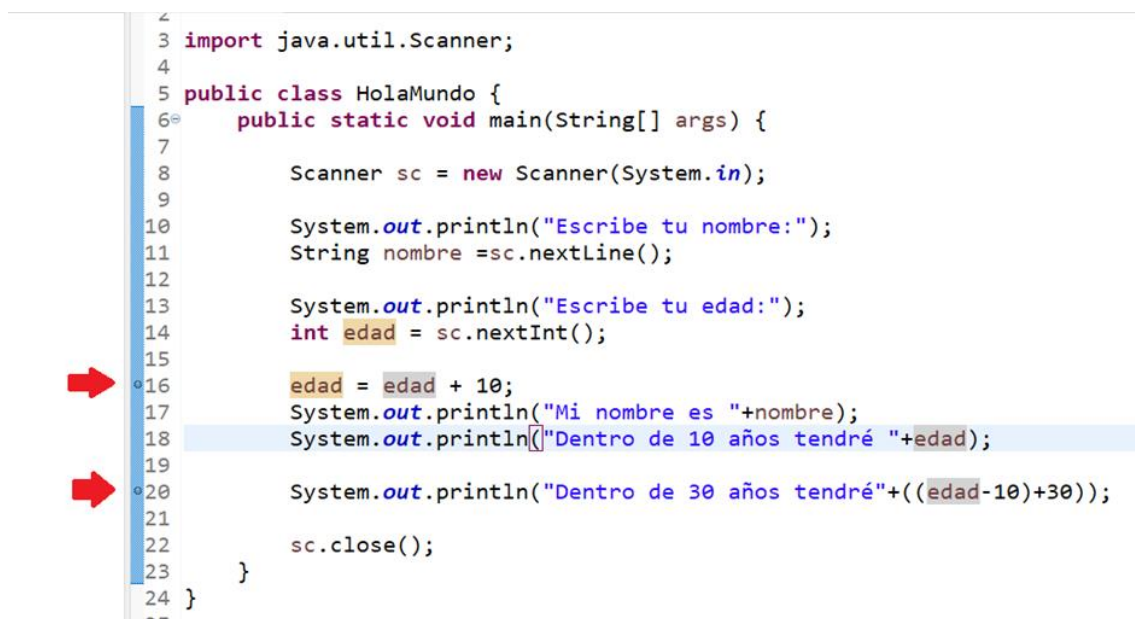



Imagen del panel superior de Eclipse.

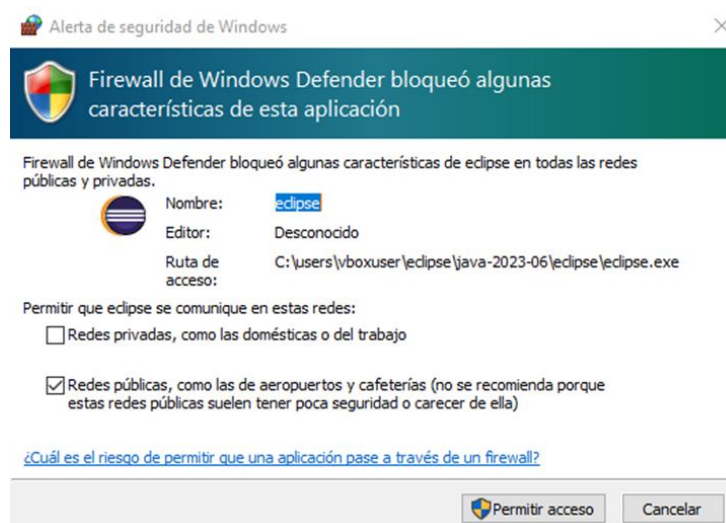
¿Cómo ejecutamos el Debugger?

Como se ha comentado anteriormente, antes de ejecutar es necesario saber qué es lo que se va a probar. Una vez que se tiene claro, localizamos donde queremos comprobar el valor de variable o de expresión y ponemos un breakpoint (punto de ruptura), el breakpoint se pone haciendo doble clic en el lateral, al lado de los números, aparecen unos puntos en cada una de las líneas de código señaladas.



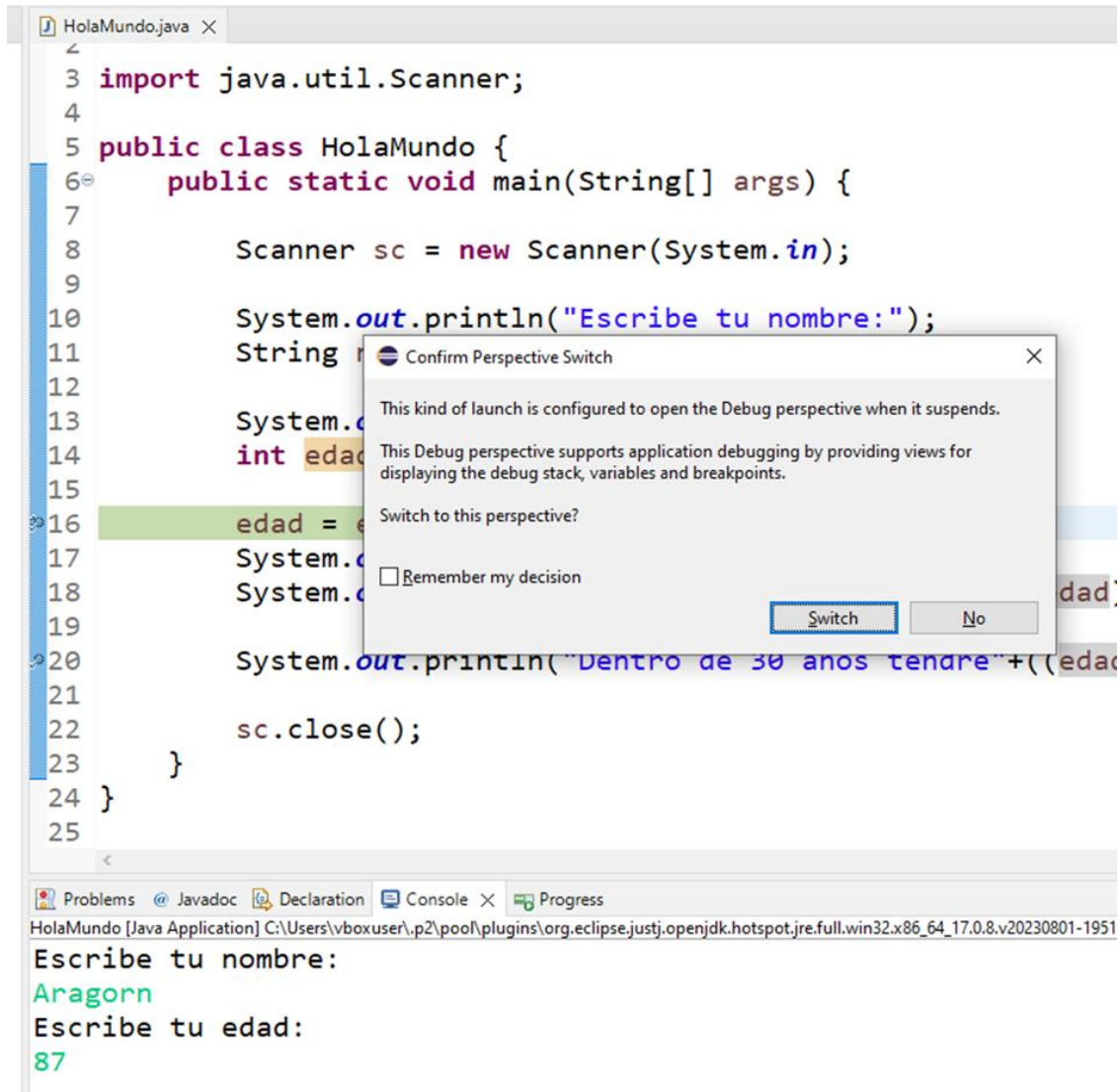
Declarando breakpoints.

Una vez marcados los breakpoints, ejecutamos en modo Debug pulsando el botón . Si es la primera vez que se ejecuta nos aparece el siguiente mensaje para crear excepción en el Firewall de Windows:



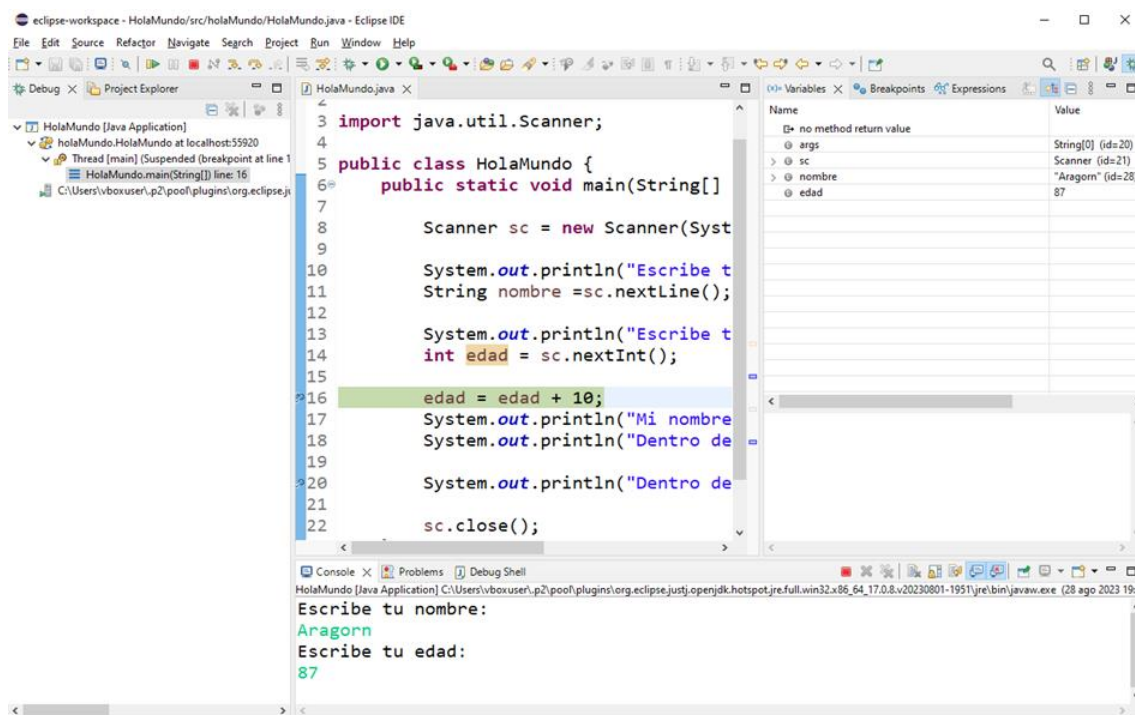
Permiso Firewall para ejecutar el modo Debug de Eclipse.

Si usamos el código del ejemplo, escribimos un nombre, escribimos una edad y se activa el modo debug al llegar al breakpoint (línea 16), aquí podemos decirle que recuerde la decisión, y si presionamos en Switch nos lleva a la vista debug de Eclipse que hemos visto anteriormente.



Cambiar a vista Debug de Eclipse

Al cambiar de vista, podemos observar varias cosas en la pantalla:



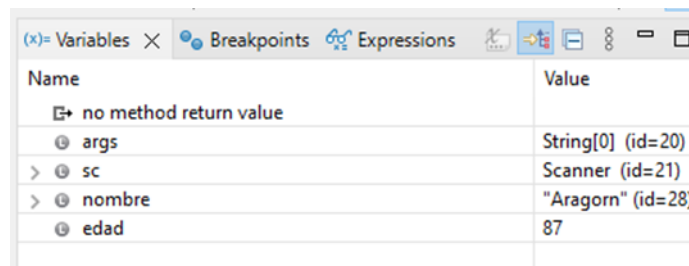
Programa JAVA en ejecución con Debug.

En la pestaña Debug aparece la ejecución, desde ahí se puede parar, pausar, o reiniciar dicha ejecución.

En la pestaña donde están los archivos .java aparece una barra verde en el código, esta barra representa por dónde va la ejecución del código. Esa barra se queda parada hasta que le digamos que avance, para ello hay tres opciones:

- **F5:** si hemos marcado un punto de ruptura, ejecutamos el modo Debug, y vamos pulsando F5, se irán ejecutando línea a línea las instrucciones del programa. Si en una línea se llama a un método que se encuentra en otra clase, automáticamente la línea verde saltará a esa clase e irá ejecutando el método conforme vayamos pulsando F5. A la vez que vamos pulsando, podemos ver en el panel de la derecha cómo el valor de las variables va cambiando.
- **F6:** Hace exactamente lo mismo que F5, salvo que no entra a los métodos que vienen predeterminados en Eclipse, cómo puede ser println, solamente entra en las clases que hemos creado nosotros.
- **F8:** Si hemos marcado más de un punto de ruptura para usar en el modo debug, al presionar F8 pasamos de un punto de ruptura a otro sin tener que ir ejecutando línea a línea cada una de las sentencias.

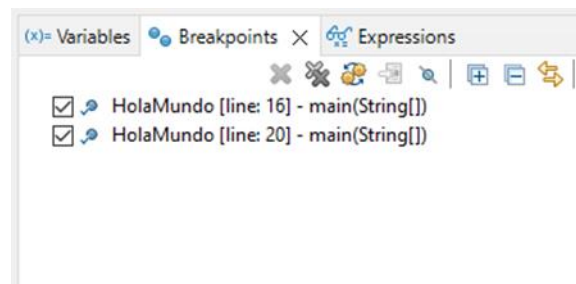
- **Variables:** en esta pestaña podemos ver el valor que van cogiendo las variables declaradas durante la ejecución del código. Si vamos presionando F5 o F6, vemos cómo cambian los valores línea a línea.



Name	Value
no method return value	
args	String[0] (id=20)
sc	Scanner (id=21)
nombre	"Aragorn" (id=28)
edad	87

Pestaña variables.

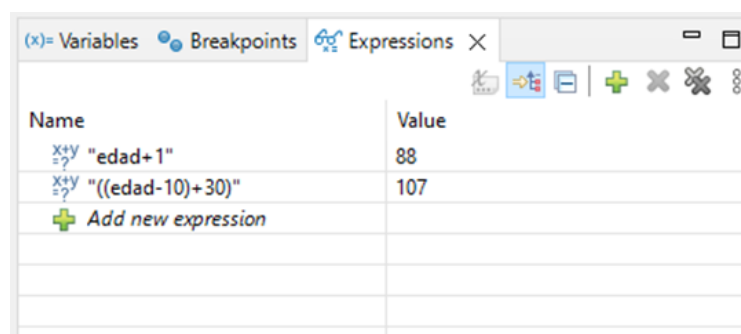
- **Breakpoints:** en esta pestaña encontraremos todos los puntos de ruptura que hay en todos los proyectos que tenemos abiertos.



Breakpoint
<input checked="" type="checkbox"/> HolaMundo [line: 16] - main(String[])
<input checked="" type="checkbox"/> HolaMundo [line: 20] - main(String[])

Pestaña Breakpoints.

- **Expressions:** sirve para evaluar expresiones del código y el valor que toman. Para que una expresión aparezca en esta pestaña, tenemos que seleccionar la expresión y darle a botón derecho Watch. La expresión aparecerá en dicha pestaña y se podrá evaluar el valor que toma durante la ejecución del código.



Name	Value
$x+y$ "edad+1"	88
$x+y$ "((edad-10)+30)"	107
Add new expression	

Pestaña Expressions.



VÍDEO DE INTERÉS

Visualiza una acción de depuración en Eclipse:



2.2 Análisis causal

En el **análisis causal** se toma información sobre los errores que se han depurado. Por tanto, se debe recoger información y para poder prevenir los próximos errores, prediciendo posibles fallos: ¿Qué se hizo mal?, ¿cómo se podría haber prevenido?, ¿cómo se podría haber detectado antes?, ¿cómo se encontró el error?, ¿por qué no se detectó antes?, ¿cuándo se cometió?, ¿Quién lo hizo?



EJEMPLO PRÁCTICO

A tu equipo de trabajo llega una aplicación software que debe ser probada con cierta urgencia, ya que el cliente espera que le sea entregada y, por consecuencia, probada, en los próximos días. ¿Con qué herramientas se probará la aplicación?

Dependerá de qué es lo que tienes que probar, de si tienes acceso al código o si el tipo de prueba es solamente funcional.

En el caso de acceder al código podemos utilizar las herramientas vistas.

Si suponemos que el código está en Java y que se usa Eclipse como IDE, podemos crear pruebas con JUnit y automatizar las mismas.

3. MEDIDAS DE CALIDAD EN EL SOFTWARE

Cuando estás desarrollando la aplicación sobre la gestión de tiendas de ropa, te surge la necesidad de poner una serie de normas a la hora de programar, nomenclatura, etc. y seguir una serie de estándares ya establecidos. Para todo ello, sabes que es necesario conocer esas normas y estándares de calidad, y saber utilizar herramientas que te ayuden en el seguimiento de los mismos.

El software desempeña un papel crítico a día de hoy en la gran mayoría de las empresas. Si además una empresa que realiza su software desea ofrecérsela a terceros, necesita de una garantía de que se está haciendo bien. En el mundo del desarrollo software como en el de otros servicios informáticos, se necesita de una supervisión constante que consta de una normativa que regulan los criterios de calidad. En este sentido, los estándares ISO15505 o CMMI proporcionan unos planteamientos estructurados para desarrollar aplicaciones software fiables.

3.1 Normas de calidad

Los distintos modelos de calidad hoy en día podrían clasificarse en función de la calidad en el producto software. En los procesos en donde se definen la creación del software o en los sistemas de gestión, las normativas se aplican en las distintas etapas del ciclo de vida de los proyectos informáticos y contribuyen a mejorar la calidad del software.



ENLACE DE INTERÉS

Visita la página donde se detallan las normas a seguir para las pruebas del software:



En la actualidad existen diversas opciones, tales como las siguientes:

- **ISO 9001:** no se basa de una forma exclusiva en la etapa del desarrollo, esta norma identifica el alcance que tendrá el software en los procesos productivos que se generan, por ejemplo, en la identificación de requisitos, en la entrega y mantenimiento.
- **ISO/IEC 9003:** norma no certificable, es una guía de buenas prácticas para definir con detalle conceptos sobre los procesos de la organización.
- **ISO/IEC 9126:** norma desarrollada entre 1991 y 2001, en donde se definen unas características y subcaracterísticas de calidad del producto, así como la calidad en su uso que van a permitir ayudar a medir la calidad del software en base a ciertos atributos de calidad.
- **ISO 25000:** de la familia de normas 25000, tiene 5 partes publicadas y establecen un modelo de calidad tanto para el software como para la evaluación.
- **ISO/IEC 12207:** Information Technology / Software Life Cycle Processes, es el estándar para los procesos de ciclo de vida del software de la organización. Es la base para ISO 15504-SPIICE (Software Process Improvement And Assurance Standards Capability Determination). Es una norma que define un modelo para todo tipo de empresas, está en continuo desarrollo. La implantación y evaluación externa para la certificación se puede realizar por etapas, de tal manera que en años posteriores irá aumentando su alcance. Esta norma evalúa la calidad software por niveles de madurez y la mejora de procesos.
- **CMMI (Capability Maturity Model Integration):** es, a nivel mundial, un requisito para acceder a la oferta de servicios software. Ofrece una guía para implementar, además del software, una estrategia de calidad y mejorar los procesos de una organización sobre el desarrollo y mantenimiento de software. Es certificable a partir de organismos privados. Es una norma dirigida a grandes empresas con requisitos de calidad muy altos cuyo desarrollo se realiza en países donde no se encuentran sus oficinas centrales y que se apoyan en pequeñas empresas que ofertan outsourcing. Su certificación verifica y puntúa en distintos niveles la organización. En 2010 hay más de 200 empresas certificadas.



¿SABÍAS QUE...?

Para alcanzar un cierto nivel de madurez en CCMI, se hace necesario bastante tiempo y esfuerzo por parte de las empresas. El valor medio para alcanzar estos niveles es: Nivel 1 al 2, 5 meses; Nivel 2 al 3, 19 meses; Nivel 3 al 4, 21 meses; Nivel 3 al 5, 25.5 meses. A partir del Nivel 3 se aumenta la complejidad en alcanzar los niveles posteriores.

3.2 Software de evaluación de calidad del código. SonarLint

Cuando una gran empresa encarga un programa, suele pedirle a la empresa que va a realizarlo que cumpla unos estándares de calidad, ejemplo de estas empresas puede ser Telefónica, Banco Santander, Inditex, etc.

Uno de esos estándares de calidad es el código. SonarLint es una plataforma abierta para gestionar la calidad del código, entre algunas de sus funciones destaca que se cumplan ciertas normas a la hora de realizar las aplicaciones:

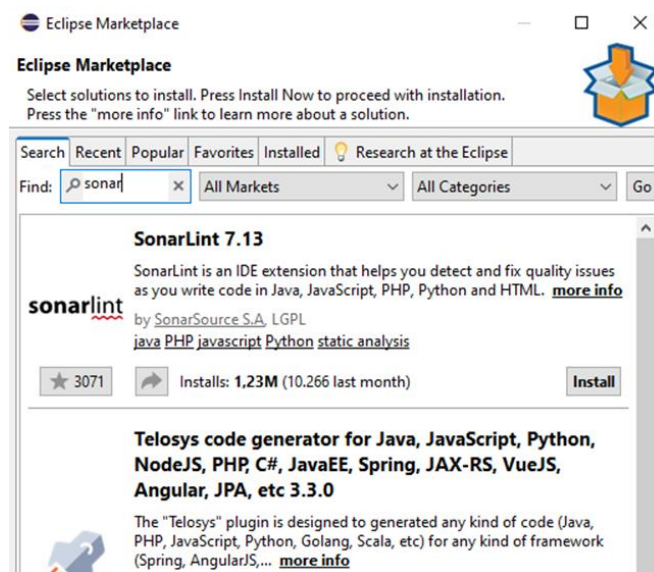
- **Estándares de codificación:** nombre de clases con primera letra en mayúscula, nombre de constantes en mayúscula, variables sin usar, variables sin nombre descriptivos.
- **Bug y errores potenciales.**
- **Duplicidad de código:** si ve que se repite código, te lo indica para qué crees un método.
- **Pruebas unitarias:** te lanza las pruebas unitarias y te da un porcentaje del código que en realidad has probado.
- **Complejidad ciclomática** (complejidad de código): detecta si el nivel es muy alto y te indica que lo bajas.
- **Control de comentarios:** si hay pocos comentarios que te indican.
- Ofrece soporte para **más de 20 lenguajes de programación.**

SonarLint en Eclipse.

Hay diferentes formas de usar el SonarLint en eclipse: en la nube, o creando un servidor.

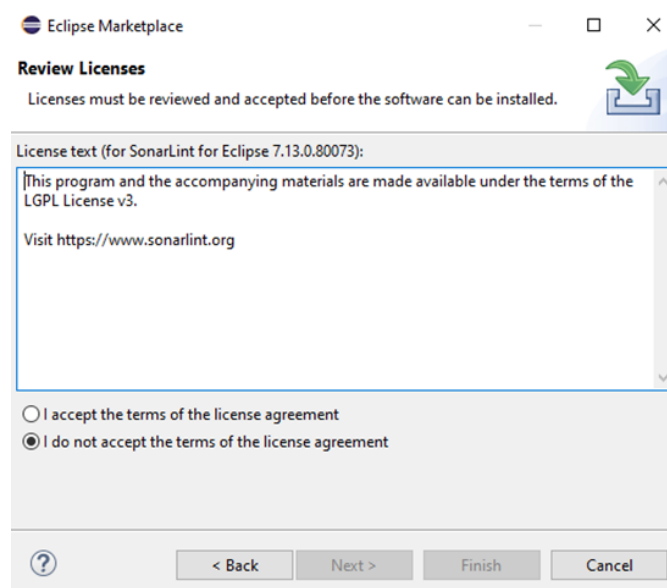
En el ejemplo que veremos a continuación se creará para usarlo en la nube:

1. Para usar el SonarLint con Eclipse, debemos instalar un plug-in en el programa. Para ello iremos a Help -> Eclipse Marketplace y en el buscador ponemos sonar. Pulsamos en "Install".



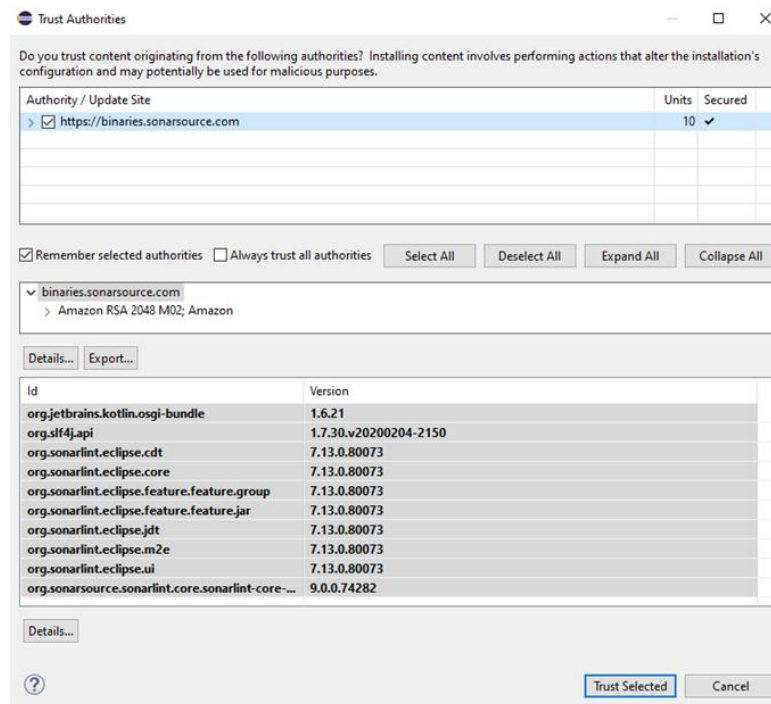
Marketplace buscando sonar.

2. Instalaremos el plug-in llamado SonarLint, tal y como se ve en las imágenes aceptando licencia.



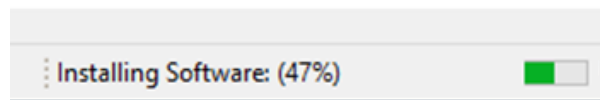
Aceptación de licencia SonarLint.

3. Seleccionamos la URL y pulsamos en “Trust Select”.

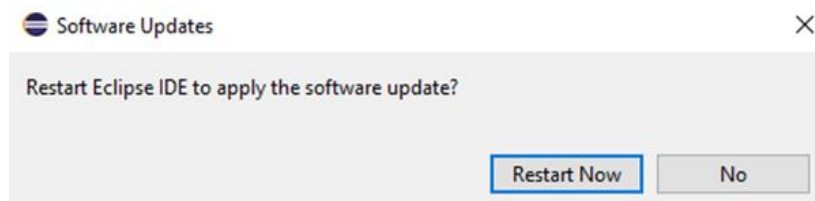


Autorización de instalación.

4. Se iniciará el proceso de instalación y al acabar, pedirá reiniciar Eclipse.

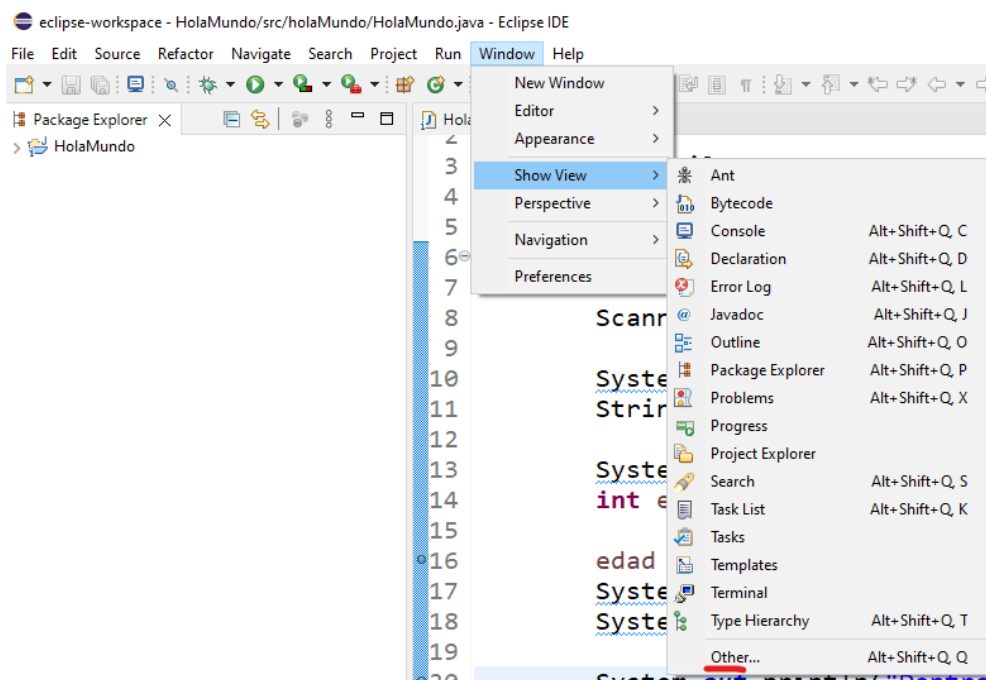


Barra de progreso de instalación.

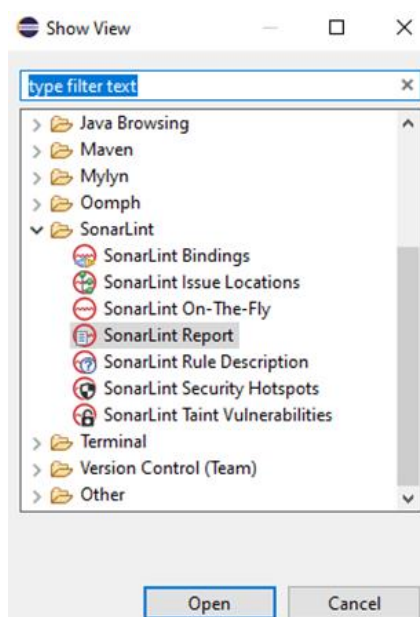


Ventana de reinicio.

5. Para acabar la configuración, abrimos una ventana para ver el reporte, para ello seleccionamos Windows → Show View → Other → SonarLint → SonarLint Report.

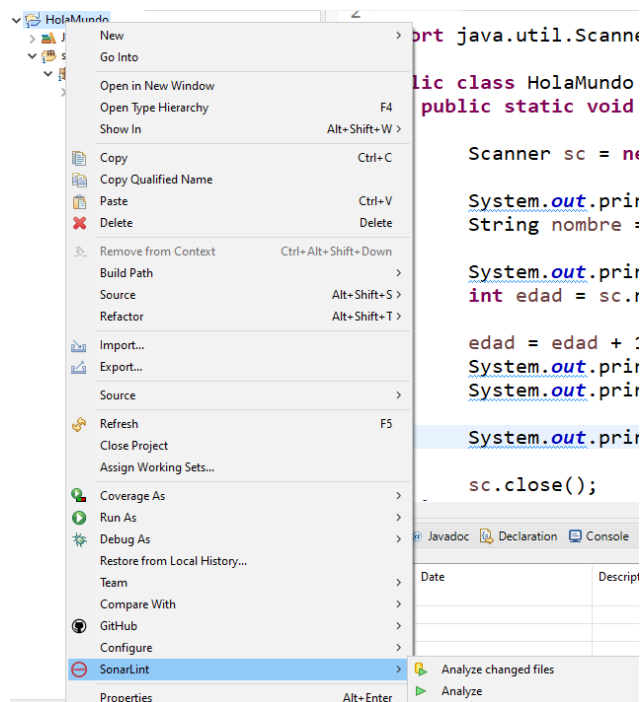


Ventana Show View para añadir paneles.



Panel con elementos SonarLint.

6. Para analizar el código, presionamos el botón derecho del ratón encima del proyecto y nos aparece como última opción SonarLint, y pulsamos en Analizar.



Analizador de código.

7. Aparece el informe en la pestaña que hemos abierto para ver los Report. Con estos resultados modificamos el código para que tenga cierta calidad establecida.

Resource	Date	Description
HolaMundo.java		Rename this package name to match the regular expression <code>^[a-z_]+(\.[a-z_][a-z0-9_]*)?\$</code> .
HolaMundo.java		Replace this use of System.out or System.err by a logger.
HolaMundo.java		Replace this use of System.out or System.err by a logger.
HolaMundo.java		Replace this use of System.out or System.err by a logger.
HolaMundo.java		Replace this use of System.out or System.err by a logger.
HolaMundo.java		Replace this use of System.out or System.err by a logger.

Reporte de analizador de código.



EJEMPLO PRÁCTICO

Al equipo de trabajo en el que estás empleado llega una aplicación software en la que se desea probar la calidad código de la aplicación. ¿Cómo procederías para medir la calidad del software?

SonarLint es una aplicación que mide la calidad del código según diversos estándares de programación, generalmente las grandes empresas exigen que se cumplan estos estándares entre en un 90-95% de todo el código de la aplicación. Instalas esta aplicación, configuras las características de calidad del software y el software te dirá el porcentaje en el que nuestra aplicación cumple los requisitos planteados.

4. JUNIT. EJEMPLO

Para el desarrollo de la aplicación sobre la gestión de tiendas de ropa, es necesario automatizar las pruebas que se han planteado en el plan de pruebas por el equipo, para ello vas a intentar automatizar el máximo posible utilizando en Eclipse el framework JUnit que te va a permitir lanzar pruebas automáticas después de cualquier modificación que se realice.

Una de las fases más importantes del ciclo de vida del software son las pruebas, ya que en ella se verifica si la aplicación cumple con los requisitos que fueron especificados por el cliente. Como se ha comprobado, existen diferentes tipos de pruebas: pruebas unitarias, que prueban los distintos módulos que forman parte del aplicativo y que indican el punto de partida para el resto de pruebas como las de integración, validación, sistema, regresión, etc.

JUnit es un framework o conjunto de herramientas open source para realizar las pruebas unitarias sobre software creado en lenguaje de programación Java. Con JUnit se organizan las pruebas y se reduce el tiempo que necesita el desarrollador o tester y así podrá centrarse en la verificación de los resultados.



VÍDEO DE INTERÉS

Visualiza algunos ejemplos de pruebas de unidad con Eclipse y JUnit:

<https://www.youtube.com/watch?v=wkXL3emg-NU>

Las pruebas con JUnit se hacen de manera organizada y controlada sobre los distintos métodos que conforman las clases en Java, haciendo una comparación entre el valor esperado y el valor que devuelven los métodos. JUnit se encarga de crear los casos y las clases de pruebas, por lo que las pruebas se realizan de manera automática. JUnit crea, por cada clase de prueba, una clase con el mismo nombre que la clase original con el sufijo Test. Es en los **asserts** o condiciones donde se comprueba el correcto funcionamiento.

Los asserts son las afirmaciones de una proposición (línea de código) en un programa o, lo que es lo mismo, una condición que debe de cumplirse, el desarrollador debe considerar que el assert siempre es verdadero. Al tener condiciones también se pueden definir precondiciones y postcondiciones de la ejecución en determinadas líneas de código, lo que se conocen como pruebas unitarias.

JUnit proporciona métodos para realizar distintas pruebas y, dependiendo de las pruebas que se desee realizar, los métodos comprueban en distintos contextos el nivel de aceptación de una prueba:

AssertEquals	Proporciona sobrecargas que permiten comprobar si un valor real coincide con el esperado.
AssertFalse	Cuando se sabe que la función siempre devuelve falso.
AssertNotNull	Se utiliza cuando existe un caso donde no se devuelva null.
AssertNotSame	Útil cuando se debe devolver un elemento de una lista que se puede usar, con el fin de determinar si ese elemento pertenece a la lista.
AssertNull	Si un método retorna null se usa este assert para comprobar su veracidad o falsedad.
Fail	Condicionales.
FailNotEquals	Hace lo mismo que assertEquals pero esperando que la prueba no sea igual.
FailNotSame	Esencialmente hace lo mismo que assertNotSame, salvo en lugar de causar un error que causa un fracaso.

Relación entre pruebas y depuración.

Se procede a crear un proyecto en lenguaje de programación Java, creando dos clases y dentro de estas se implementarán 4 métodos que serán sobre los que se realizarán las pruebas unitarias, a través del entorno de desarrollo Eclipse.

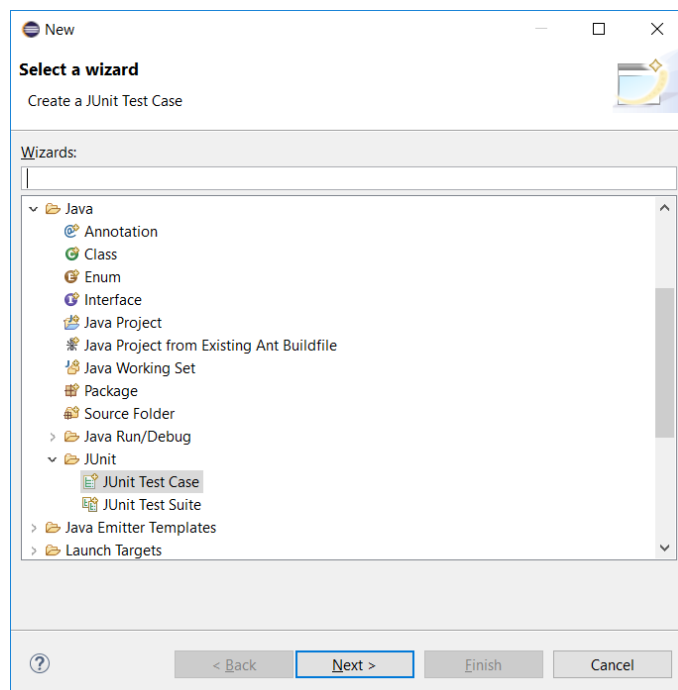
4.1 Crear proyecto en Eclipse

Como primer paso procederemos a crear un proyecto Java en Eclipse.

Una vez que tenemos un proyecto en JAVA con una serie de clases creadas y métodos para probar, procedemos a crear clases en JUnit para la realización de pruebas.

En el ejemplo que vamos a seguir lo vamos a hacer al contrario, primero crearemos la clase JUnit y después crearemos la clase con el código que queremos probar.

Para comenzar, abrimos el IDE Eclipse, creamos un proyecto JAVA, una vez creado el proyecto, creamos un paquete, dentro de ese paquete creamos una clase JUnit, presionando con el botón derecho en el proyecto New → Other → Java → JUnit → JUnit Test Case. Pulsar Next.



Crear proyecto JUnit en Eclipse 1.

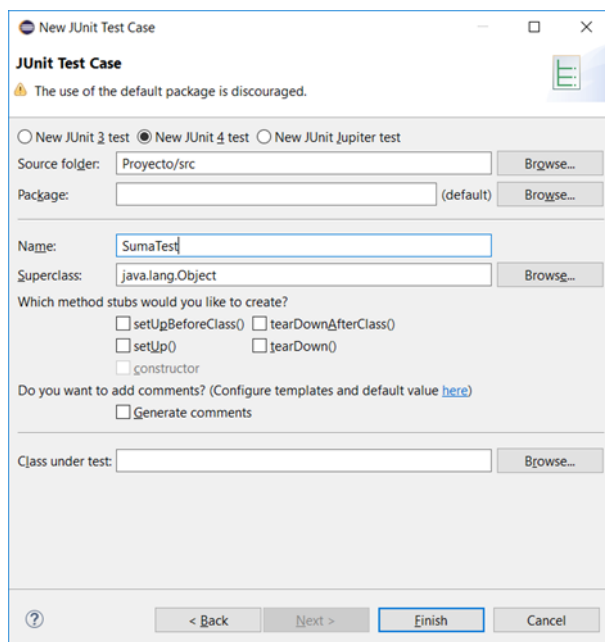


VÍDEO DE INTERÉS

Conoce un tutorial de JUnit en Eclipse:

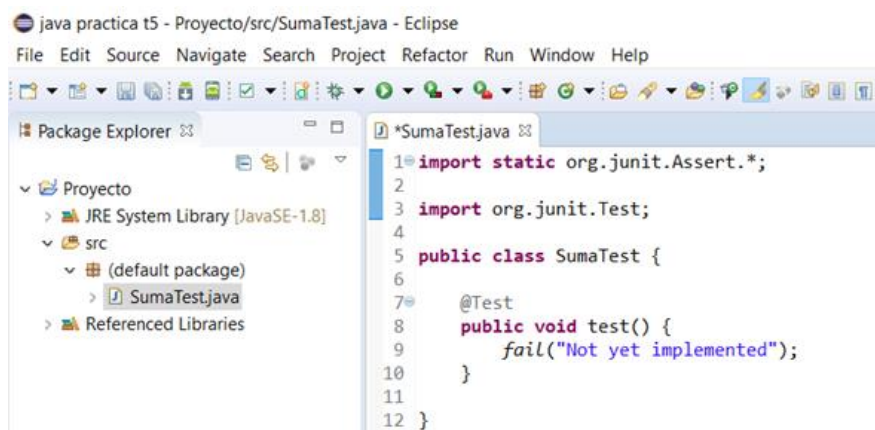


En la siguiente interfaz, justo en el nombre (Name) escribir SumaTest, para realizar unas pruebas unitarias sencillas sobre una suma. Pulsar Finish:



Crear proyecto JUnit en Eclipse 2.

En este preciso momento se ha creado, de forma automática, una clase llamada SumaTest.java que contiene las siguientes líneas de código:

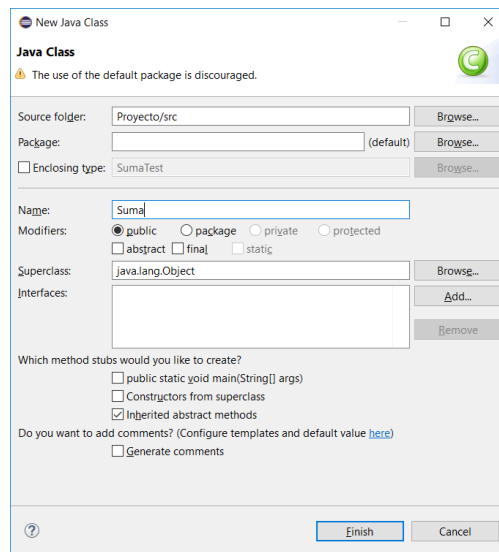


Crear proyecto JUnit en Eclipse 3.

Además, se ha añadido una librería al proyecto llamada JUnit 5.

4.2 Crear clase Suma

La segunda parte de este ejemplo consiste en la creación de la clase Suma que va a ser la encargada de realizar la operación suma. Aunque en estos momentos no se tenga mucho conocimiento sobre los conceptos de clase y objeto (programación orientada a objetos), no es necesario saber más allá de lo propuesto en esta guía de ejemplo. Para crear la clase Suma, el procedimiento es botón derecho en el proyecto New → Class → en Name escribir Suma:



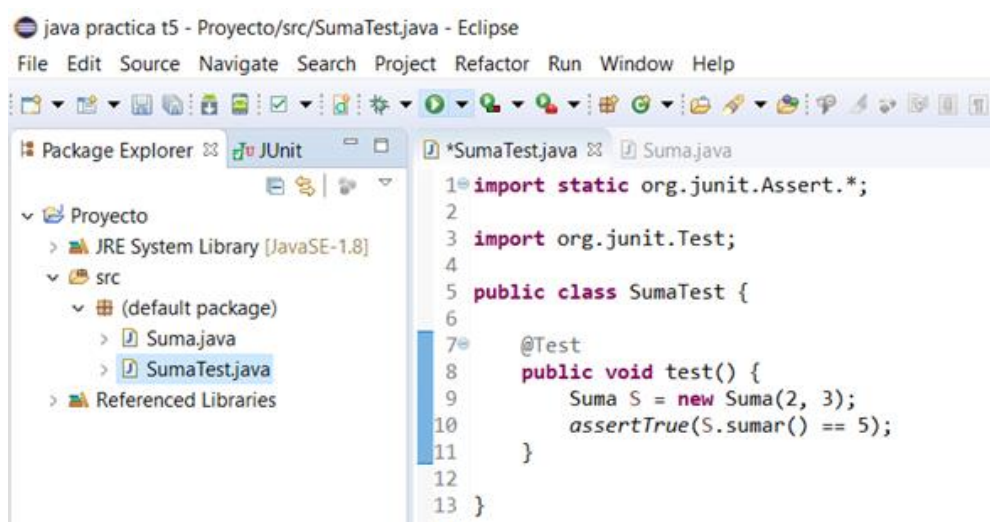
Crear clase para pruebas JUnit en Eclipse 1.

De forma automática se crea una clase (vacía) que se llama Suma.java dentro de la cual habrá que añadirle el siguiente código:

```
public class Suma {  
  
    int num1;  
  
    int num2;  
  
    public Suma(int num1, int num2) {  
        super();  
        this.num1 = num1;  
        this.num2 = num2;  
    }  
  
    public int sumar(int n1, int n2) {  
        int resultado = n1 + n2;  
        return resultado;  
    }  
}
```

Es importante que el código respete todos y cada uno de los caracteres, ya que cualquier error no permitirá su ejecución. Grosso modo, esta clase Suma.java presenta 2 variables (num1 y num2), por tanto, permitirá sumar 2 números de tipo entero (int). Posee 2 métodos: **Suma**, para la creación de objetos y **sumar**, que retorna la suma de num1+num2. Se insiste en la idea de que no es preciso conocer Java en estos momentos para realizar pruebas unitarias, simplemente seguir esta guía para realizar las pruebas unitarias con JUnit.

Una vez implementada la clase Suma.java, ya es posible realizar los casos de prueba en la clase SumaTest.java. Para ello, se procede a visualizar esta clase y programar en ella las siguientes líneas de código:

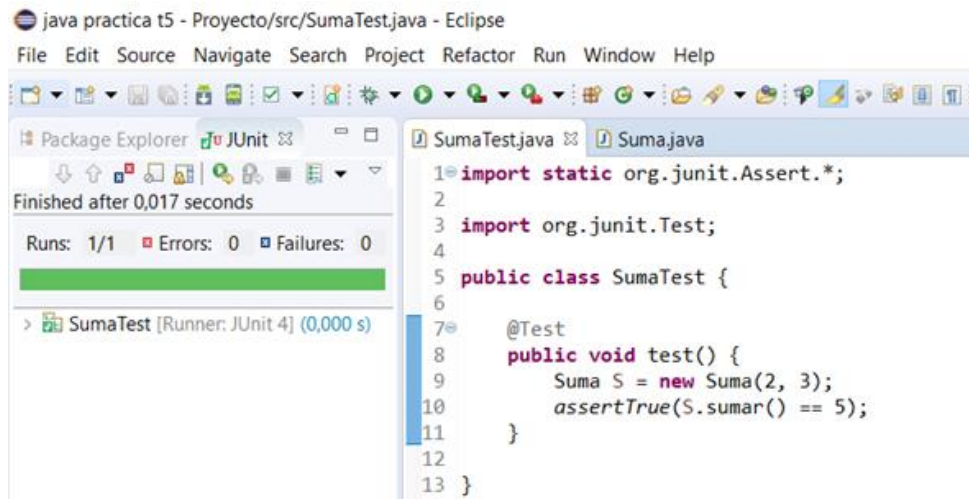


Crear clase para pruebas JUnit en Eclipse 2.

Como se puede comprobar, únicamente se han añadido 2 líneas:

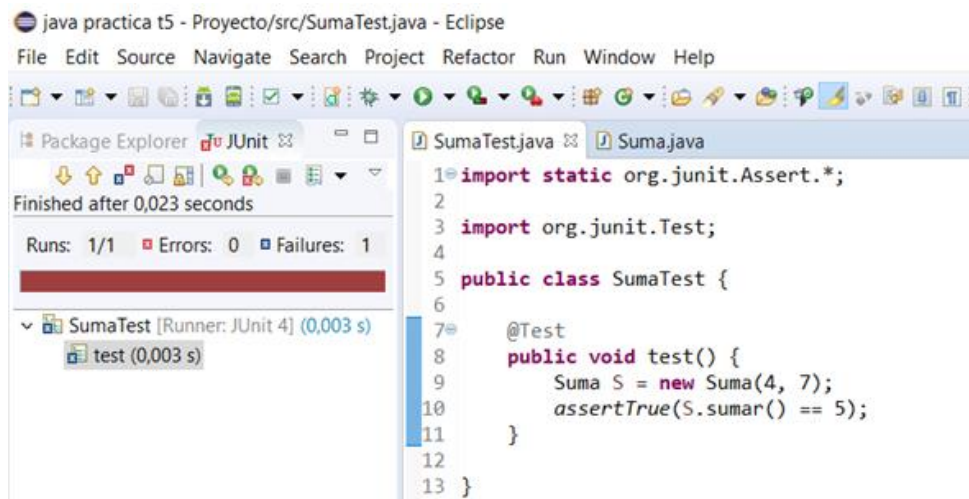
```
Suma S = new Suma(2, 3);
assertTrue(S.sumar() == 5);
```

La primera de ellas va a permitir crear un objeto de la clase Suma, otorgando un valor "2" a num1 y un valor "3" a num2. La segunda línea va a realizar una llamada al método sumar, con esos valores, estimando que la suma va a dar como resultado el valor "5". Con estos valores de prueba introducidos, ¿qué resultado se obtiene de forma automática con JUnit? Bastará con ejecutar la clase SumaTest.java (botón play) y observar resultado esperado válido (verde):



Crear clase para pruebas JUnit en Eclipse 3.

Cambiando cualquier valor no esperado, por ejemplo, si el caso de prueba consiste en sumar 4 y 7, manteniendo el resultado anterior, la validación JUnit dictará que el resultado obtenido no es el esperado, ya que $5 \neq 11$ (marrón).



Crear clase para pruebas JUnit en Eclipse 4.

RESUMEN FINAL

En esta unidad se ha destacado la importancia de las pruebas en el desarrollo del software. En la fase de pruebas, es importante establecer un plan de pruebas y una estrategia para ser llevado a cabo. El equipo de pruebas será el encargado de realizar las diferentes pruebas a realizar, tales como las pruebas de unidad, de integración, de sistema y de validación. Será en el entorno del cliente donde se realizarán las de aceptación, momento en el cual el cliente aceptará o no el software desarrollado, en función del cumplimiento de los requisitos establecidos al comienzo del mismo.

Como proceso interesante, en esta unidad se destaca la gran labor que tiene la automatización de las pruebas a través de las herramientas que la soportan. Gracias a ella, los casos de prueba, procedimientos de prueba y componentes de prueba se van a poder ejecutar y validar de forma automática. Existen una serie de pruebas que podrían encuadrarse dentro de las pruebas de sistema, llamadas pruebas de rendimiento, que van a permitir llevar las aplicaciones al límite de su carga y medir el comportamiento en base a unos parámetros de rendimiento, como por ejemplo el consumo de memoria, el procesamiento y los tiempos de ejecución, entre otros. Estas pruebas de rendimiento no serían posibles sin la automatización, ya que se llevan a cabo a través de software específico capaz de ejecutar acciones estresantes para el aplicativo y el servidor que lo contiene. Por ejemplo, mil usuarios concurrentes realizando un login o un logout durante una hora, o doscientos usuarios concurrentes realizando transferencias bancarias sin descanso en un periodo constante de más de 5 horas.

Por último, hay que destacar la relación existente entre la calidad del software y las pruebas, tomando como ejemplo una serie de normativas de calidad que proporcionan características y subcaracterísticas de calidad medibles (ISO 9126, ISO 25000, etc.). En este sentido, cuanto mayor probado esté un software, mayor número de defectos se habrán encontrado, con lo que se habrá trabajado en solventarlos y el producto estará (casi) libre de errores. Este hecho le otorgará un nivel de calidad bastante aceptable.

También se ha podido ver la configuración y automatización de pruebas con el framework JUnit, uno de los más usados en el mercado actualmente.