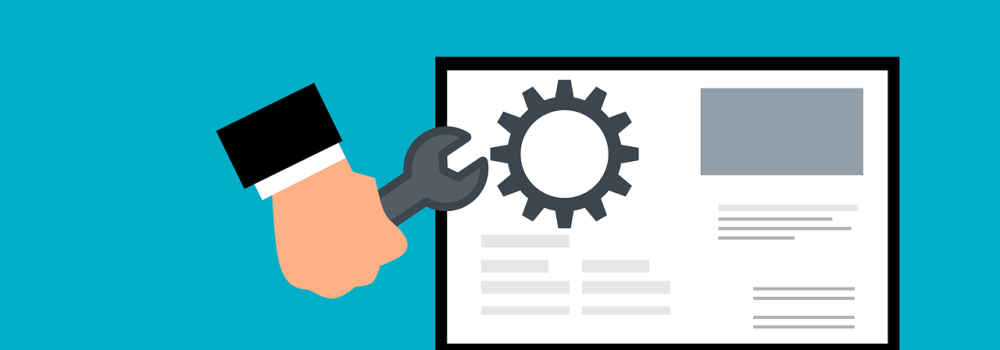
Pruebas y Depuración en Entornos de Desarrollo



**Autor: Ignacio Castillón Salguero**  
**Docente: Victor Manuel Navarro Camino**  
**Centro: Cesur Malaga Este**  
**Ciclo: Desarrollo de Aplicaciones Multiplataformas**  
**Fecha Entrega: 2 de mayo de 2025**

Índice

**1. RA 3 – DISEÑO Y REALIZACIÓN DE PRUEBAS**

a. Identificar los diferentes tipos de pruebas. Definir casos de prueba. Identificado las herramientas de depuración y prueba de aplicaciones ofrecidas por el entorno de desarrollo.  
b. Utilizar herramientas de depuración para definir puntos de ruptura y seguimiento. Utilizar las herramientas de depuración para examinar y modificar el comportamiento de un programa en tiempo de ejecución.  
c. Documentar el plan de pruebas.  
d. Efectuar pruebas unitarias de clases y funciones. Efectuar pruebas de integración, de sistema y de aceptación.

e. Implementar pruebas automatizadas. Documentar las incidencias detectadas.  
f. Aplicar normas de calidad a los procedimientos de desarrollo de software. Realizar medidas de calidad sobre el software desarrollado.

**2. RA 4 – OPTIMIZACIÓN Y DOCUMENTACIÓN**

f. Documentar el código fuente mediante comentarios.  
g. Utilizar herramientas del entorno de desarrollo para documentar los procesos, datos y eventos.  
h. Utilizar herramientas del entorno de desarrollo para documentar las clases.

**3. Bibliografía / Webgrafía**

**4. Anexos**

Tipos de pruebas

Las pruebas de software son un componente esencial del desarrollo, ya que permiten asegurar que la aplicación funcione correctamente en todas sus etapas. Existen distintos tipos de pruebas, cada una con un propósito específico dentro del ciclo de vida del software.

Pruebas unitarias  
 Este tipo de pruebas se centra en verificar el correcto funcionamiento de los módulos o unidades individuales del software, como funciones, métodos o clases. Se realizan de manera aislada para asegurarse de que cada componente por separado hace exactamente lo que se espera de él. Generalmente, las pruebas unitarias son implementadas por los propios desarrolladores y suelen ser automatizadas. Son fundamentales para detectar errores en etapas tempranas del desarrollo y asegurar una base sólida sobre la cual construir funcionalidades más complejas.

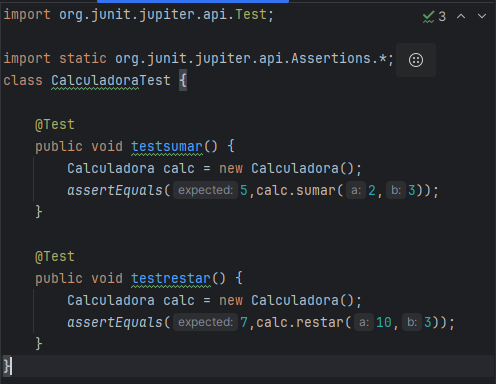
Pruebas de integración  
 Una vez que las unidades individuales han sido probadas de forma aislada, las pruebas de integración se encargan de evaluar cómo interactúan entre sí. El objetivo principal es comprobar que los módulos o componentes del sistema funcionan correctamente cuando se comunican entre ellos. Esto incluye la verificación de interfaces, intercambio de datos, sincronización y flujos de trabajo entre diferentes partes del software. Estas pruebas ayudan a identificar problemas que no son evidentes en pruebas unitarias, como errores de comunicación o incompatibilidades.

Pruebas de regresión  
 Este tipo de pruebas se realiza después de hacer cambios en el código, ya sea para corregir errores o agregar nuevas funcionalidades. El propósito de las pruebas de regresión es asegurar que esas modificaciones no hayan afectado negativamente otras partes del sistema que antes funcionaban correctamente. Las pruebas de regresión se ejecutan con frecuencia y, en muchos casos, están automatizadas para garantizar una verificación continua durante el desarrollo. Son cruciales para mantener la estabilidad del sistema a lo largo del tiempo.

Pruebas de validación  
 Las pruebas de validación permiten comprobar si el software cumple con los requisitos especificados al inicio del proyecto. A diferencia de las pruebas unitarias o de integración, estas pruebas evalúan el comportamiento del sistema como un todo, considerando tanto la funcionalidad como el rendimiento, la seguridad, la usabilidad y otros factores no funcionales. El objetivo es asegurarse de que el producto final se alinea con las expectativas del cliente o usuario final.

Pruebas de sistema y rendimiento  
 En esta etapa, se realiza una evaluación completa del sistema en su conjunto. Las pruebas de sistema buscan comprobar que todos los componentes se han integrado correctamente y que el software realiza las funciones previstas de manera efectiva. Por otro lado, las pruebas de rendimiento se enfocan en medir la capacidad del sistema bajo ciertas condiciones, analizando aspectos como la velocidad de respuesta, el tiempo de carga, el uso de recursos y la estabilidad frente a cargas de trabajo variables. Estas pruebas son fundamentales para garantizar una experiencia de usuario fluida y un comportamiento confiable bajo distintas circunstancias.

Pruebas de aceptación  
 Finalmente, las pruebas de aceptación son realizadas por el cliente o usuario final en un entorno lo más cercano posible al de producción. Su objetivo es verificar si el software realmente cumple con los requisitos del cliente y si está listo para su implementación. Estas pruebas son decisivas para la aprobación del producto final, ya que representan la validación directa por parte de quien utilizará el sistema. Si las pruebas de aceptación son satisfactorias, el software puede considerarse listo para su despliegue.



Casos de prueba

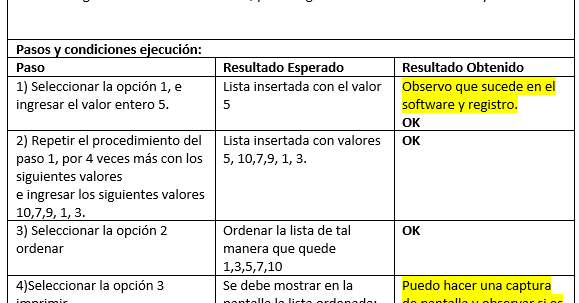
Un caso de prueba es una secuencia de pasos diseñada específicamente para validar el comportamiento de una funcionalidad o de un requisito dentro de un sistema de software. Su principal objetivo es comprobar si una parte del sistema funciona como se espera en un escenario determinado, utilizando entradas conocidas y verificando los resultados obtenidos en comparación con los resultados esperados.

Cada caso de prueba describe de manera detallada qué se debe hacer, cómo se debe hacer, con qué datos de entrada y cuál debería ser el resultado correcto si el sistema está funcionando correctamente. Esta estructura detallada permite que las pruebas sean reproducibles, claras y fácilmente evaluables, tanto por el desarrollador como por otros miembros del equipo.

En el caso de que el resultado obtenido no coincida con el esperado, el caso de prueba cumple una función aún más importante: ayuda a localizar el origen del problema. Al tener documentados todos los pasos realizados y los datos utilizados, se puede revisar paso a paso qué parte del proceso ha fallado y por qué. Esto permite identificar errores de forma precisa y facilita su corrección, lo cual mejora considerablemente la calidad y estabilidad del software.

Además, los casos de prueba pueden formar parte de una batería de pruebas más amplia, donde se incluyen distintos escenarios: tanto los esperados como los casos límite o situaciones inesperadas. Esto permite tener una visión más completa del comportamiento del sistema ante diferentes condiciones, lo que contribuye a reducir riesgos y aumentar la fiabilidad del software.

En resumen, un caso de prueba no solo sirve para comprobar que algo funciona, sino que también es una herramienta de documentación, diagnóstico y control de calidad dentro del proceso de desarrollo.



Herramientas de depuración

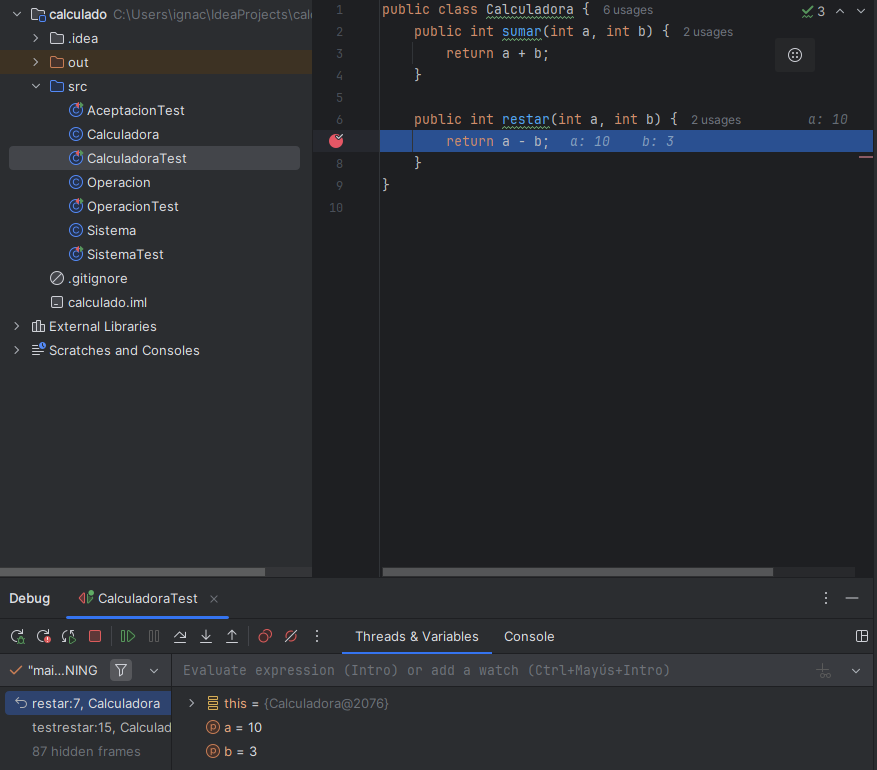
La depuración es un proceso fundamental en el desarrollo de software. Se entiende por depuración al conjunto de técnicas y herramientas utilizadas para identificar, aislar y corregir errores en el código fuente de una aplicación. Estos errores, también conocidos como bugs, pueden afectar tanto el comportamiento como el rendimiento del sistema, por lo que detectarlos y solucionarlos de manera eficiente es clave para asegurar un software confiable y funcional.

Depurar no solo significa encontrar un fallo visible, sino también comprender su causa raíz. Muchas veces un error visible en la ejecución tiene su origen en una parte del código completamente distinta. La depuración permite analizar en detalle cómo se comporta el programa en tiempo de ejecución, qué valores toman las variables, cómo fluyen los datos y en qué punto exacto se produce el fallo. Gracias a esto, los desarrolladores pueden implementar soluciones precisas y evitar arreglos superficiales que puedan causar más errores en el futuro.

Entre los principales beneficios de la depuración se encuentran la mejora de la calidad del software, la prevención de problemas más graves durante la etapa de producción y una mayor eficiencia en el proceso de desarrollo. Al reducir la cantidad de errores que llegan al usuario final, se incrementa también la satisfacción del cliente y se disminuyen los costos de mantenimiento.

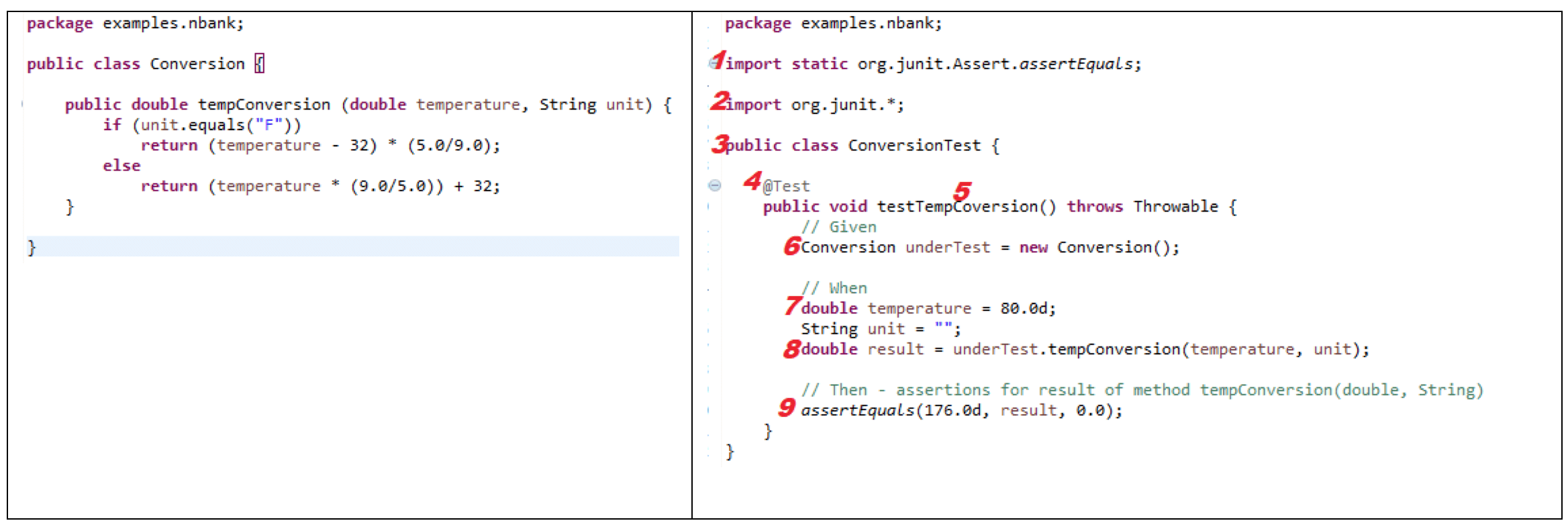
Existen diversas herramientas de depuración que facilitan este trabajo. Una de las más comunes es el uso de compiladores o entornos de desarrollo integrados (IDE) que permiten ejecutar el programa paso a paso, observar el valor actual de las variables, establecer puntos de interrupción (breakpoints), modificar el flujo de ejecución y analizar el comportamiento del sistema en distintas condiciones. Estas funciones permiten al desarrollador tener un control más profundo sobre lo que está ocurriendo en tiempo real dentro de la aplicación.

Los depuradores son especialmente útiles cuando se necesita entender cómo actúa un programa ante una entrada específica o en una determinada secuencia de operaciones. Al poder detener la ejecución en un punto concreto, se puede estudiar con precisión qué está pasando en ese instante del proceso. Esto resulta clave para resolver errores complejos, especialmente en aplicaciones grandes o con múltiples interacciones entre componentes.



Herramientas de prueba

* Las pruebas de software son una etapa crítica en el proceso de desarrollo de aplicaciones, ya que permiten evaluar y verificar que el sistema cumple correctamente con las especificaciones y requisitos definidos. Su propósito principal es detectar fallos o comportamientos inesperados antes de que el producto llegue al usuario final, minimizando así riesgos, reduciendo costos de mantenimiento y aumentando la calidad general del software.
* Entre los principales beneficios de las pruebas de software se encuentran la prevención de errores, la mejora del rendimiento, la validación del cumplimiento de los requisitos funcionales y no funcionales, y el aumento de la confianza tanto del equipo de desarrollo como del cliente en el producto final.
* Tipos de pruebas de software
* Las pruebas de software pueden clasificarse en dos grandes categorías: pruebas funcionales y pruebas no funcionales.
* Pruebas funcionales  
   Este tipo de pruebas se centra en verificar si el software cumple con los requisitos funcionales establecidos. En otras palabras, se valida si el sistema hace lo que se espera que haga. Entre las pruebas funcionales más comunes se encuentran:
* Pruebas unitarias: Evalúan módulos individuales del software, como funciones o métodos, de forma aislada. Suelen ser implementadas por los desarrolladores y se utilizan para comprobar que cada parte del código funciona correctamente por sí sola.
* Pruebas de integración: Verifican la correcta comunicación entre módulos o componentes del sistema. Se centran en evaluar cómo interactúan las distintas partes del software entre sí.
* Pruebas de sistema: Validan el comportamiento del sistema como un todo, comprobando que todos los componentes funcionan de manera conjunta tal como se especificó en los requisitos.
* Pruebas de aceptación: Estas se realizan normalmente al final del desarrollo, y su objetivo es que el cliente o usuario final valide si el sistema satisface sus expectativas y requisitos. Es la última etapa antes de poner el software en producción.
* Pruebas no funcionales  
   A diferencia de las funcionales, las pruebas no funcionales no se centran en lo que el sistema hace, sino en cómo lo hace. Evalúan aspectos relacionados con el rendimiento, la estabilidad, la seguridad y la experiencia del usuario, entre otros. Algunas pruebas no funcionales importantes incluyen:
* Pruebas de rendimiento: Analizan la velocidad de respuesta del sistema, su estabilidad bajo diferentes condiciones y el consumo de recursos como CPU y memoria.
* Pruebas de carga: Evalúan el comportamiento del sistema cuando es utilizado por múltiples usuarios simultáneamente. Permiten identificar cuellos de botella o fallos que aparecen bajo condiciones de alta demanda.
* Pruebas de seguridad: Se enfocan en detectar vulnerabilidades y riesgos que puedan comprometer la integridad, confidencialidad o disponibilidad del sistema.
* Pruebas de usabilidad: Analizan qué tan fácil y agradable resulta para el usuario interactuar con la aplicación. Evalúan la interfaz, la navegación, la comprensión de las funciones, entre otros aspectos.
* Herramientas de prueba populares
* En el mercado existen múltiples herramientas que facilitan la automatización y ejecución de pruebas. Algunas de las más utilizadas son:
* JUnit: Es una de las herramientas más populares para realizar pruebas unitarias en aplicaciones desarrolladas con Java. Permite automatizar la validación del código de forma sencilla y efectiva.
* Selenium: Es una herramienta ampliamente utilizada para automatizar pruebas funcionales en aplicaciones web. Permite simular la interacción del usuario con la interfaz del sitio, como hacer clic, llenar formularios y verificar resultados en distintas condiciones y navegadores.
* Estas herramientas, junto con otras como TestNG, Postman, Apache JMeter, entre muchas más, permiten aumentar la cobertura de pruebas, reducir errores humanos y acelerar el proceso de validación del software.



Utilizar herramientas de depuración para definir puntos de ruptura y seguimiento

Los puntos de ruptura son herramientas fundamentales en el proceso de depuración de software. Se trata de indicadores o marcadores que se colocan en líneas específicas del código con el objetivo de que el proceso de ejecución del programa se detenga en ese punto durante la depuración. Esta detención permite al desarrollador inspeccionar el estado del programa en ese momento exacto, lo cual es crucial para identificar el origen de los errores o problemas que puedan estar ocurriendo.

Cuando se ejecuta un programa en modo de depuración, el flujo normal de la ejecución se interrumpe automáticamente al alcanzar un punto de ruptura. En ese momento, el desarrollador tiene la oportunidad de examinar el valor de las variables, el contenido de las estructuras de datos, la pila de llamadas, las condiciones del sistema y otros aspectos importantes. Esto permite entender cómo se está comportando el programa, por qué ocurre un error en ese momento específico y cómo solucionarlo.

Además de servir como una herramienta para encontrar errores, los puntos de ruptura también pueden ayudar a validar el comportamiento del software bajo condiciones específicas, a comprobar la lógica detrás de las decisiones del código o incluso a verificar cómo se están ejecutando las funciones a medida que se pasa de un módulo a otro.

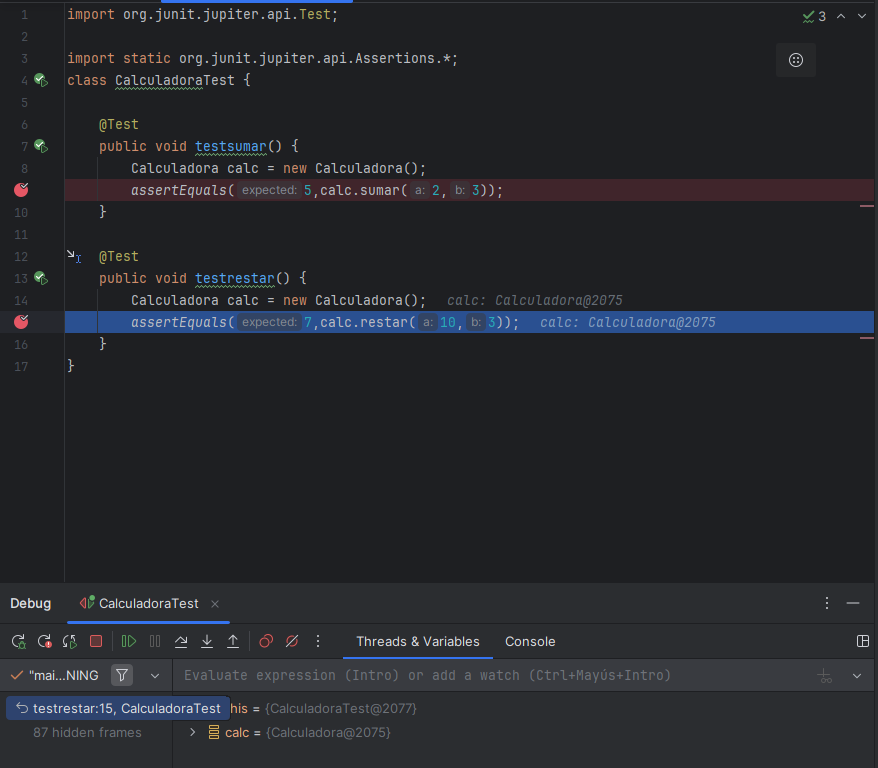
Los puntos de ruptura pueden ser de diferentes tipos, como:

Puntos de ruptura condicionales: Estos se activan solo cuando se cumplen ciertas condiciones. Por ejemplo, el punto de ruptura podría activarse solo si el valor de una variable supera un umbral determinado.

Puntos de ruptura en el método: Permiten detener la ejecución en la entrada o salida de un método, lo que es útil cuando se desea inspeccionar cómo se comportan las funciones dentro de un flujo de ejecución más amplio.

Puntos de ruptura por excepción: Se activan cuando se lanza una excepción, lo que permite detener el programa justo en el momento en que se produce un error, facilitando el análisis y resolución de problemas.

El uso adecuado de los puntos de ruptura facilita la depuración de programas complejos, optimizando la búsqueda de errores y mejorando la calidad del software desarrollado.



Utilizar las herramientas de depuración para examinar y modificar el comportamiento de un programa en tiempo de ejecución

Las herramientas de depuración no solo son fundamentales para verificar y evaluar el comportamiento del código, sino que también ofrecen funcionalidades avanzadas que permiten modificar el estado del programa mientras se está depurando. Esta capacidad de manipular variables y modificar su valor en tiempo real es extremadamente útil, ya que proporciona una mayor flexibilidad para probar diferentes condiciones y escenarios sin necesidad de modificar el código fuente directamente.

Durante el proceso de depuración, los desarrolladores pueden utilizar herramientas de depuración como puntos de ruptura, inspección de variables, y ejecución paso a paso. Sin embargo, una de las características más potentes es la posibilidad de modificar las variables en ejecución. Esto significa que se pueden cambiar los valores de las variables en cualquier momento, sin tener que reiniciar el programa o modificar el código de forma permanente.

La capacidad de modificar variables durante la depuración permite a los desarrolladores probar rápidamente cómo se comporta el sistema bajo diferentes condiciones, lo que facilita la identificación y resolución de errores. Por ejemplo, si un error está ocurriendo debido a un valor incorrecto de una variable, el desarrollador puede modificar esa variable durante la depuración para verificar si el sistema se comporta correctamente con un valor diferente, sin necesidad de alterar el código base.

Algunas de las principales ventajas de poder modificar variables durante la depuración incluyen:

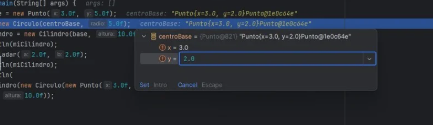
Pruebas rápidas de escenarios alternativos: Si se sospecha que un error se debe a un valor específico de una variable, modificar su valor durante la depuración permite comprobar rápidamente si el sistema se comporta como se espera con un valor diferente.

Aislamiento de errores: Si el problema está relacionado con el valor de una variable en un punto específico del código, la posibilidad de cambiar ese valor ayuda a aislar el error de manera más precisa, haciendo que la depuración sea más eficiente.

Simulación de condiciones excepcionales: Modificar variables también es útil para simular condiciones especiales o excepcionales que pueden ser difíciles de reproducir de forma natural. Esto es útil cuando se desea probar cómo se comporta el sistema en situaciones extremas o inusuales sin necesidad de esperar a que se presenten en tiempo real.

Mejora en la optimización del rendimiento: Cambiar valores de variables en tiempo de ejecución también puede ayudar a identificar cuellos de botella o problemas de rendimiento, ya que permite probar diferentes configuraciones sin tener que modificar el código permanentemente.

Las herramientas de depuración modernas, como los depuradores integrados en entornos de desarrollo (IDE) como IntelliJ IDEA, Visual Studio, Eclipse, entre otros, proporcionan interfaces gráficas o de línea de comandos que permiten hacer todo esto de manera intuitiva. Los desarrolladores pueden ver el valor actual de las variables, modificarlas en tiempo real, y continuar la ejecución del programa para observar los efectos de esos cambios.



Documentar el plan de pruebas.

* Un plan de pruebas es un documento formal que define con claridad cómo se evaluará un sistema de software para asegurar que cumpla con los requisitos funcionales y no funcionales especificados. Este documento orienta todo el proceso de pruebas, proporcionando una hoja de ruta para el equipo de calidad, desarrolladores y otros involucrados en el ciclo de vida del software.
* El plan de pruebas busca detectar errores antes de que el sistema sea liberado, asegurar la calidad del producto y minimizar riesgos. También ayuda a establecer una base clara para tomar decisiones sobre la liberación del software.

Estructura de un plan de pruebas

1. Nombre del módulo o componente

* Especifica el área del sistema o el componente que se va a probar. Puede ser una clase, un módulo funcional (como autenticación o carrito de compras), una API, o incluso una interfaz gráfica.
* Ejemplo: Módulo de registro de usuarios

2. Objetivo de la prueba

* Define claramente qué se pretende validar o verificar con la ejecución de las pruebas. Esto puede ser comprobar el correcto funcionamiento, asegurar la integración entre componentes o validar el rendimiento bajo condiciones específicas.
* Ejemplo: Verificar que el sistema permita registrar nuevos usuarios con datos válidos y rechace datos incompletos o inválidos.

3. Funcionalidades a probar

* Detalla las funciones o características específicas del sistema que se someterán a pruebas. Deben estar directamente relacionadas con los requisitos del software.
* Ejemplo:
* Validación de campos obligatorios
* Verificación del formato del correo electrónico
* Confirmación de contraseña
* Registro exitoso con datos válidos

4. Tipos de prueba

* Se especifican los distintos niveles de prueba que se aplicarán para cubrir las distintas fases del desarrollo y asegurar la estabilidad del sistema.
* Ejemplo:
* **Pruebas unitarias**: Validan el correcto funcionamiento de métodos individuales.
* **Pruebas de integración**: Aseguran que los módulos interactúan correctamente entre sí.
* **Pruebas de sistema**: Evalúan el comportamiento del sistema como un todo.
* **Pruebas de aceptación**: Verifican que el software cumple con lo acordado con el cliente o usuario final.

5. Casos de prueba

* Son los escenarios específicos que se van a ejecutar. Cada caso debe incluir:
* Identificador único
* Descripción del caso
* Datos de entrada
* Pasos detallados para ejecutar la prueba
* Resultado esperado

6. Herramientas utilizadas

* Describe los entornos, plataformas o aplicaciones empleadas para llevar a cabo las pruebas. Pueden incluir herramientas de automatización, entornos de desarrollo, servidores o navegadores específicos.
* Ejemplo:
* JUnit para pruebas unitarias
* Selenium para pruebas de interfaz web automatizadas
* Postman para pruebas de APIs REST
* IntelliJ IDEA como entorno de desarrollo
* Google Chrome como navegador de pruebas

7. Responsables

* Indica quiénes son los encargados de ejecutar cada fase de pruebas. Puede incluir roles como QA testers, desarrolladores, líderes de proyecto o incluso usuarios finales en pruebas de aceptación.

8. Resultado esperado

* Establece los criterios para que una prueba se considere satisfactoria. Puede incluir valores esperados, redirecciones, mensajes de éxito o comportamientos específicos del sistema.

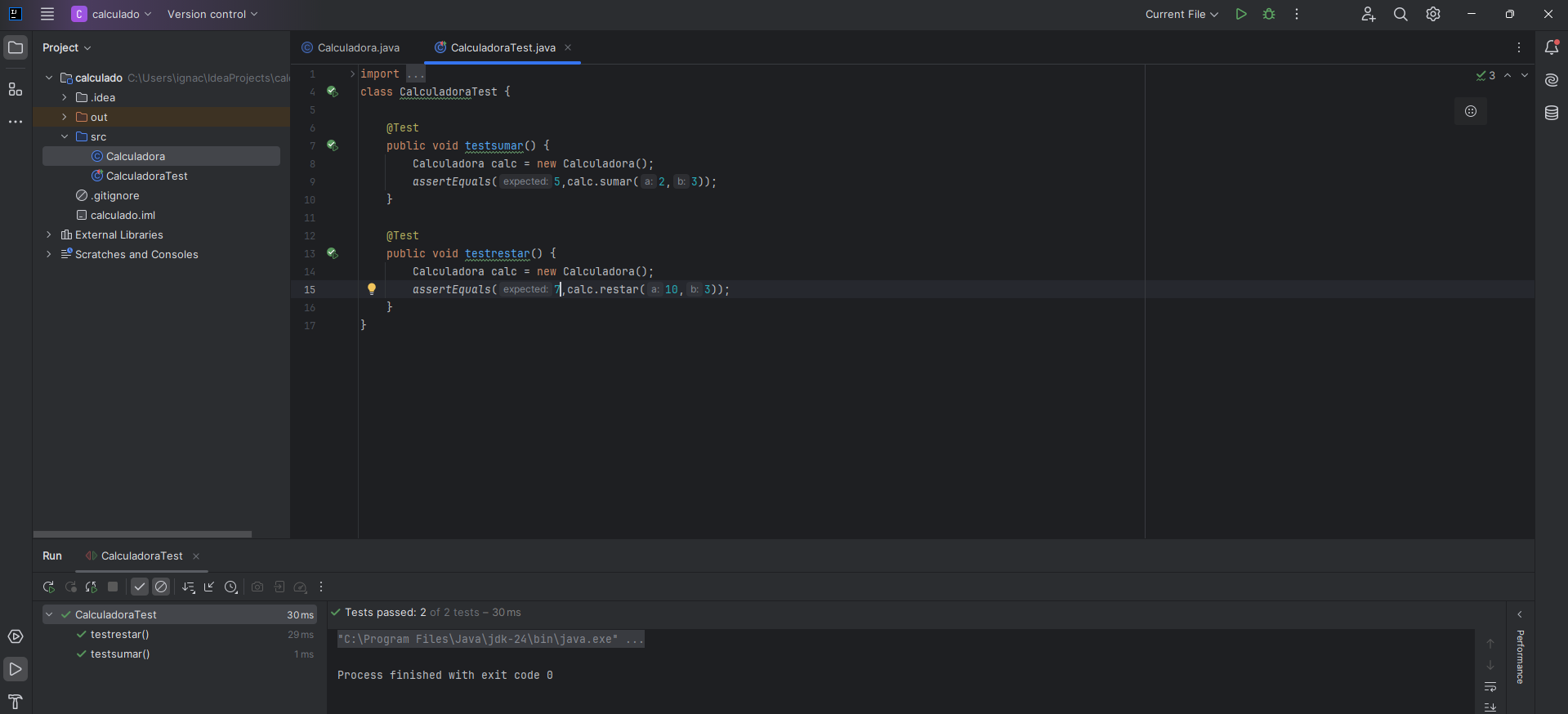
Ejemplo:

* El usuario debe ser registrado correctamente y recibir un correo de confirmación.
* Los errores deben mostrarse claramente si los datos ingresados no son válidos.

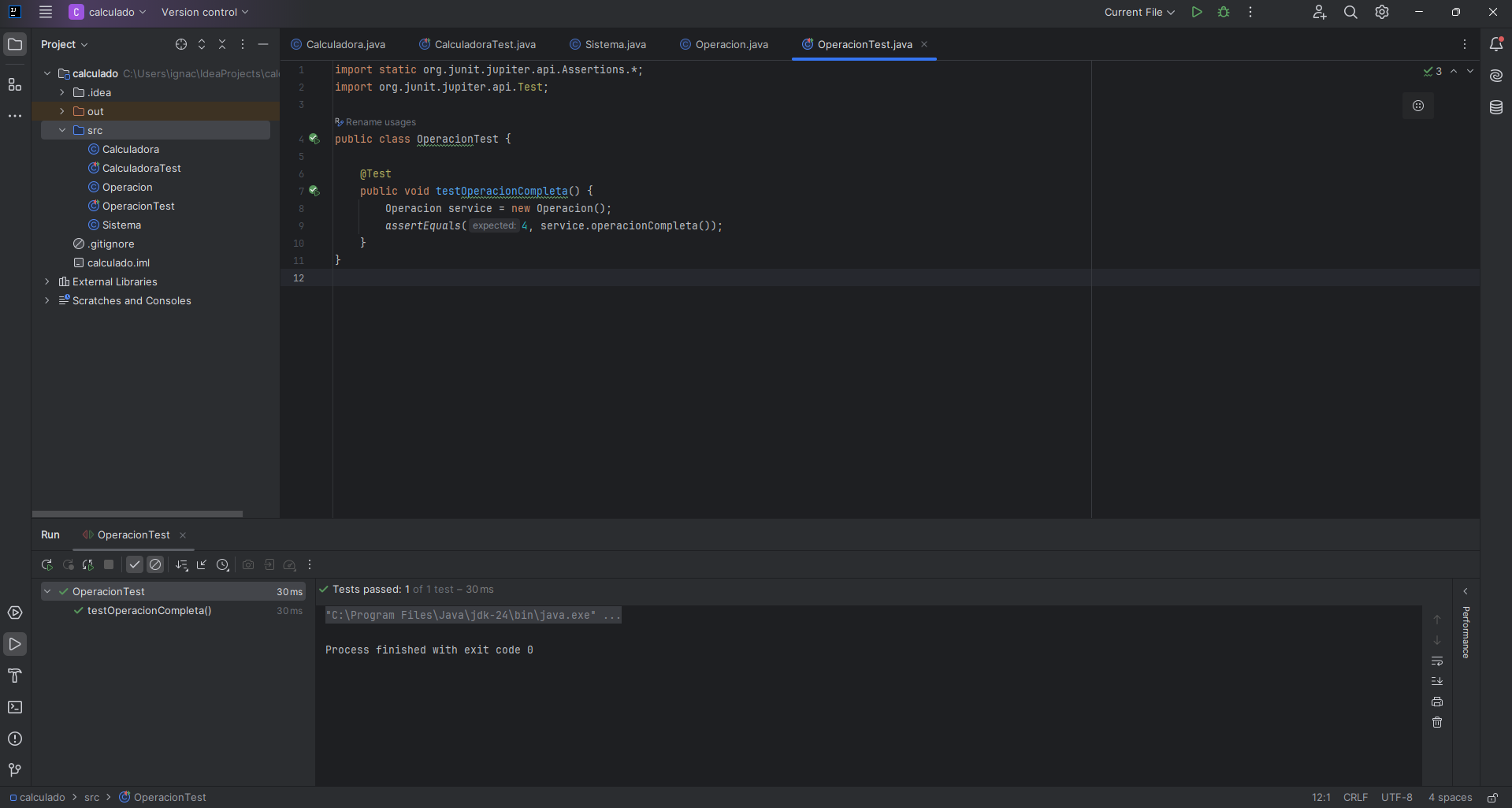


Efectuar pruebas unitarias de clases y funciones. Efectuar pruebas de integración, de sistema y de aceptación.

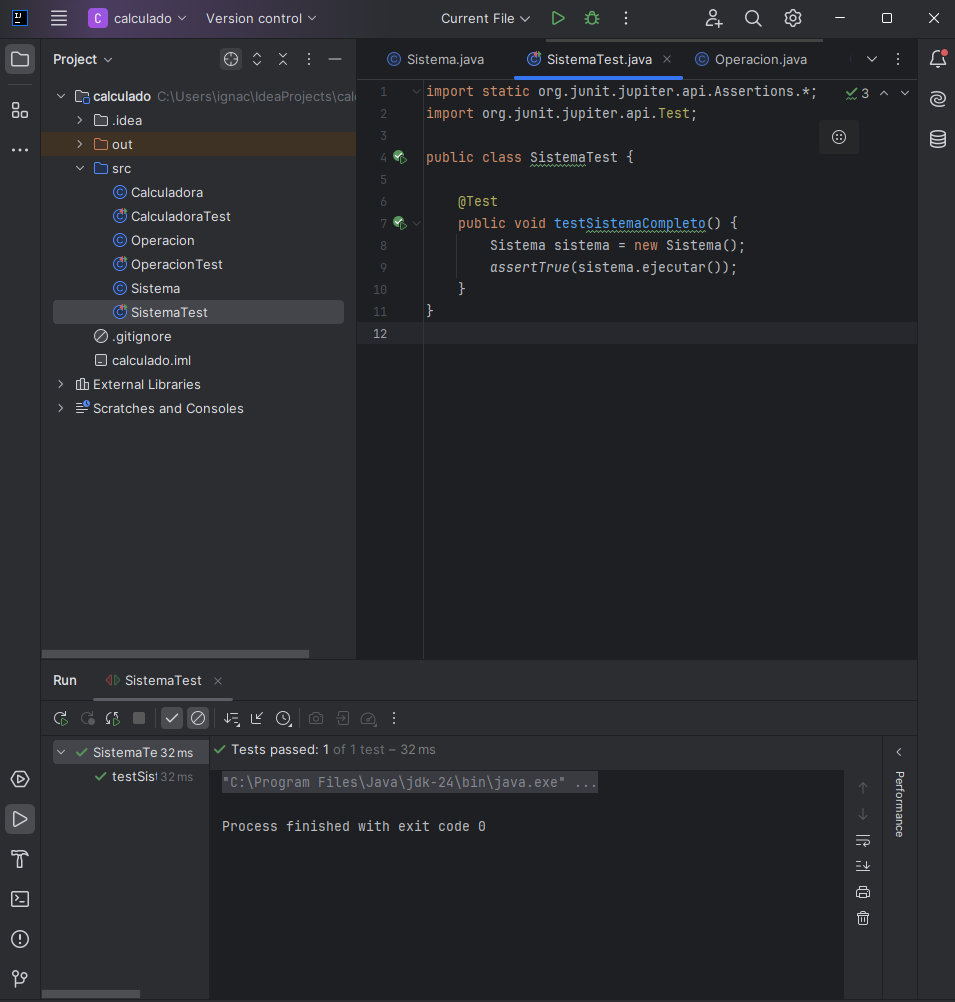
Prueba unitaria



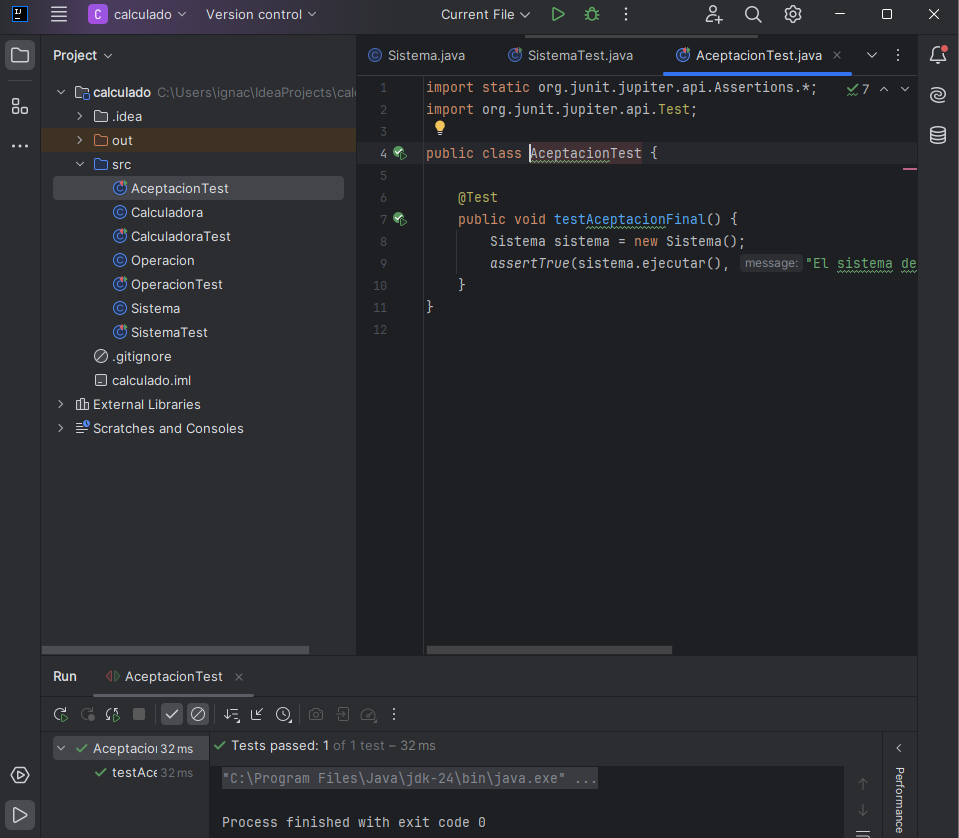
Prueba de integración



Prueba de sistema



Prueba de aceptación



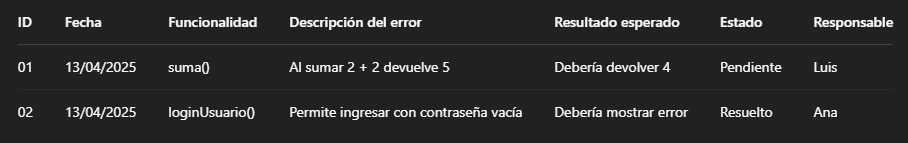
Implementar pruebas automáticas. Documentar las incidencias detectadas.

Las pruebas automatizadas son una parte muy importante del proceso de desarrollo de software hoy en día. Se trata de utilizar herramientas que nos permiten ejecutar pruebas de forma automática, sin tener que hacerlo manualmente una y otra vez. Estas herramientas no solo se encargan de correr las pruebas, sino que también analizan los resultados y los comparan con lo que se espera que suceda. Esto es especialmente útil para tareas repetitivas, ya que elimina el error humano y ahorra mucho tiempo.

Una de las grandes ventajas de automatizar las pruebas es que podemos hacerlas constantemente, cada vez que se actualiza el código, sin que alguien tenga que estar pendiente todo el tiempo. Así es mucho más fácil detectar errores rápidamente y asegurarse de que todo sigue funcionando como debería.

Cuando una de estas pruebas automatizadas detecta un error o falla en el sistema, lo ideal es registrarlo de inmediato. Para que esa información sea útil, es importante anotarla de forma organizada. Normalmente, se le asigna una identificación única al error para poder distinguirlo fácilmente. También se registra la fecha en que ocurrió, el módulo o parte del código que se estaba probando y cuál fue el resultado que se obtuvo. Toda esta información se guarda en una herramienta de gestión o en una tabla, lo que facilita mucho el seguimiento y la solución de los problemas.

Contar con este tipo de registro también nos ayuda a tener una visión más clara de qué partes del sistema fallan con más frecuencia, o si hay errores que se repiten. Eso permite tomar decisiones más informadas y mejorar el desarrollo del software con el tiempo.



Aplicar normas de calidad a los procedimientos de desarrollo de software. Realizar medidas de calidad sobre el software desarrollado.

Seguir reglas, estándares y buenas prácticas durante el desarrollo de software es esencial para asegurar que el producto final sea de alta calidad, fácil de mantener y libre de errores críticos. Estas prácticas no solo mejoran el funcionamiento técnico del sistema, sino que también facilitan la colaboración entre desarrolladores y el mantenimiento a largo plazo.

Una de las prácticas fundamentales es el cumplimiento de normas de codificación. Esto implica escribir el código con una estructura clara, ordenada y coherente, respetando las guías de estilo del lenguaje de programación que se esté utilizando. Esto permite que cualquier miembro del equipo pueda entender rápidamente el funcionamiento del sistema y trabajar sobre una base común.

El control de versiones también desempeña un papel clave. A través de herramientas como Git, es posible llevar un registro detallado de todos los cambios realizados en el código, identificar a los responsables de cada modificación y revertir errores si es necesario. Una buena gestión de versiones también implica el uso de ramas para separar funcionalidades, correcciones o pruebas, lo que favorece un desarrollo más seguro y organizado.

Otra práctica importante es la revisión de código. Consiste en que un desarrollador revise el código escrito por otro antes de integrarlo al proyecto principal. Este proceso permite detectar errores o posibles mejoras desde una segunda perspectiva, previniendo problemas antes de que lleguen a producción.

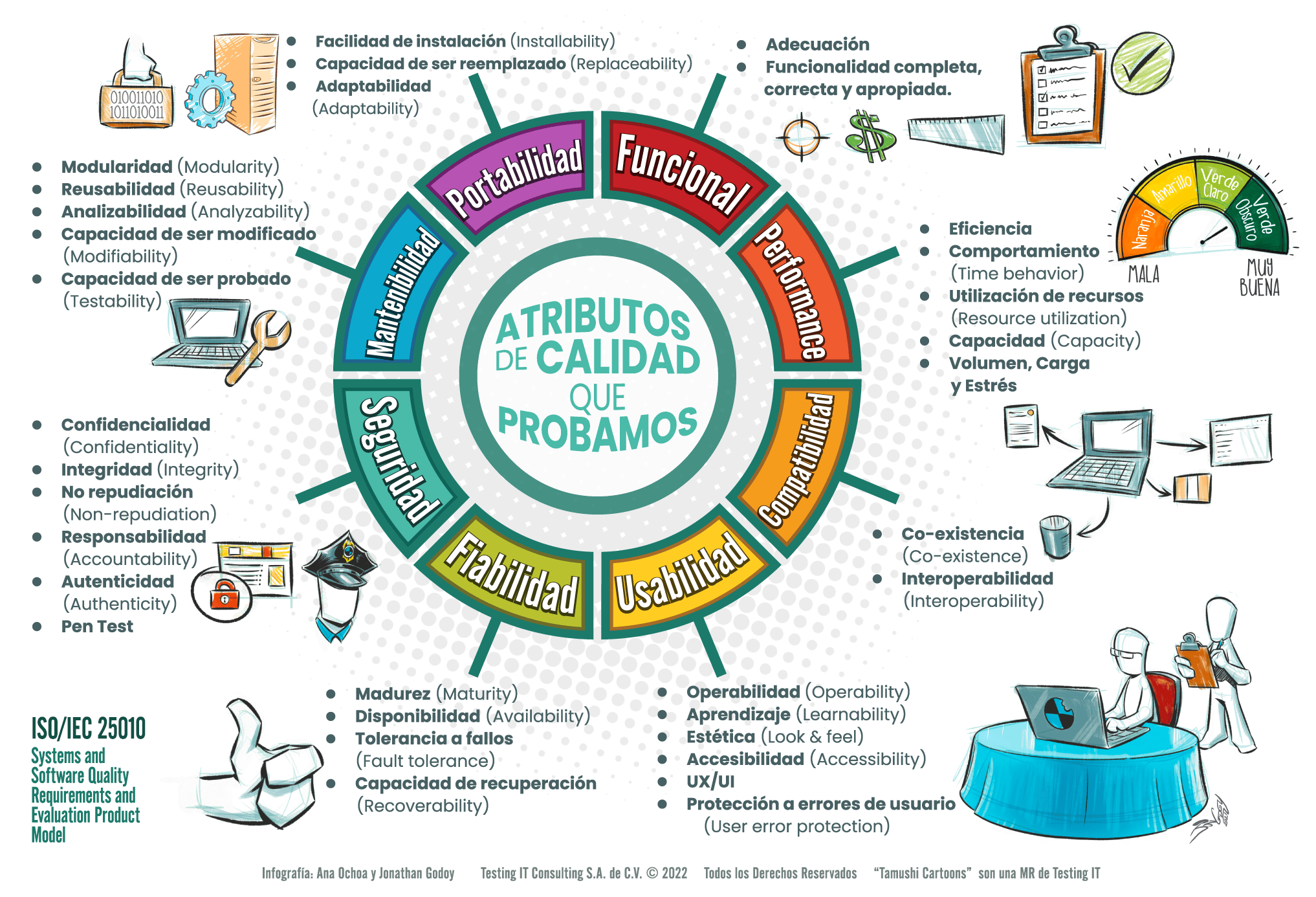
La documentación técnica también es una parte esencial del desarrollo de software de calidad. Documentar adecuadamente clases, métodos y funciones ayuda a explicar el propósito y funcionamiento del código, especialmente en secciones complejas. Esto facilita tanto el mantenimiento como la incorporación de nuevos desarrolladores al equipo.

La automatización de pruebas es otra práctica clave. Implementar pruebas automáticas permite verificar de forma constante que el código funciona correctamente y que los cambios futuros no afecten el comportamiento esperado del sistema. Estas pruebas se pueden integrar dentro de un proceso de integración continua.

La integración continua consiste en automatizar la ejecución de pruebas y el despliegue del software cada vez que se introduce un cambio en el código. Esta práctica reduce significativamente la posibilidad de errores en producción y acelera el ritmo de desarrollo.

En proyectos de mayor envergadura, también es recomendable seguir normas internacionales de calidad como las normas ISO o modelos de madurez como CMMI. Estas normas proporcionan un marco estructurado para asegurar la calidad del software desde su planificación hasta su entrega final.

Aplicar todas estas prácticas en conjunto contribuye significativamente a garantizar que el software sea confiable, estable, escalable y mantenible a lo largo del tiempo.

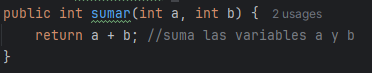


Documentar el código mediante comentarios

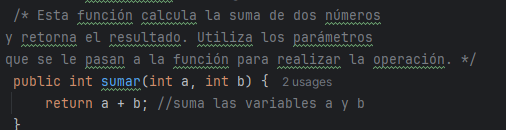
La documentación del código mediante comentarios es una práctica esencial en el desarrollo de software, ya que permite explicar el propósito y el funcionamiento de bloques de código, lo que facilita su comprensión tanto para otros desarrolladores como para el propio autor en el futuro. Los comentarios actúan como anotaciones que explican la lógica detrás del código, los posibles casos especiales o decisiones tomadas durante su desarrollo. Esta técnica no solo mejora la legibilidad del código, sino que también ayuda en la identificación de posibles mejoras, errores o áreas que necesitan ser optimizadas.

Existen diferentes tipos de comentarios que se pueden utilizar:

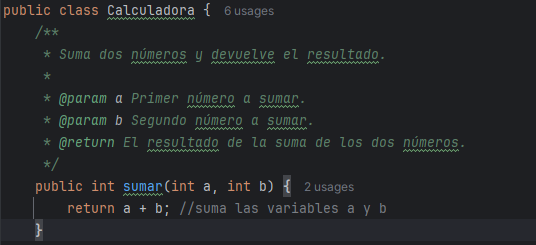
1. **Comentarios de una sola línea**: Se utilizan para explicar una parte muy específica del código o para describir rápidamente una acción que realiza una línea de código.



1. **Comentarios multilínea**: Se utilizan cuando se necesita proporcionar una explicación más detallada o cuando se está documentando una sección completa del código.



1. **Comentarios Javadoc**: Son comentarios especiales utilizados en Java para generar documentación automática. Este tipo de comentarios se coloca antes de las clases, métodos o variables y está diseñado para describir sus parámetros, valores de retorno y otras características importantes.



Utilizar herramientas del entorno de desarrollo para documentar los procesos, datos y eventos.

Estas herramientas permiten que el código sea más fácil de entender, mantener y mejorar. Al documentar procesos, datos y eventos, se ayuda no solo al desarrollador que escribe el código, sino también a cualquier otra persona que en el futuro tenga que revisarlo, modificarlo o ampliarlo.

Documentar procesos, datos y eventos se refiere a:

Procesos: Se refiere a las acciones o secuencias que sigue el software para realizar una tarea específica, como registrar un usuario o calcular un total. Documentar un proceso implica explicar qué hace, cómo lo hace y en qué orden.

Datos: Aquí hablamos de las variables, estructuras, objetos o cualquier tipo de información que se manipula en el programa. Documentarlos significa describir qué representa cada dato, de dónde viene, y para qué se utiliza.

Eventos: Son situaciones que ocurren durante la ejecución de un programa, como hacer clic en un botón, recibir un mensaje o cambiar el estado de una aplicación. Documentar eventos sirve para saber cómo se manejan y qué consecuencias tienen en el flujo del programa.

Estas documentaciones se realizan de distintas formas:

Mediante comentarios en el código: Se pueden usar comentarios para explicar líneas específicas de código, bloques lógicos o comportamientos.

Con documentación automática (como Javadoc en Java): Se escribe en un formato especial encima de clases o métodos y el entorno puede generar una documentación formal a partir de eso.

Herramientas visuales del IDE: Algunos entornos permiten crear diagramas, adjuntar notas, ver el flujo de eventos en tiempo real o registrar cambios a medida que se codifica.

Además, la documentación del código tiene distintas ventajas como:

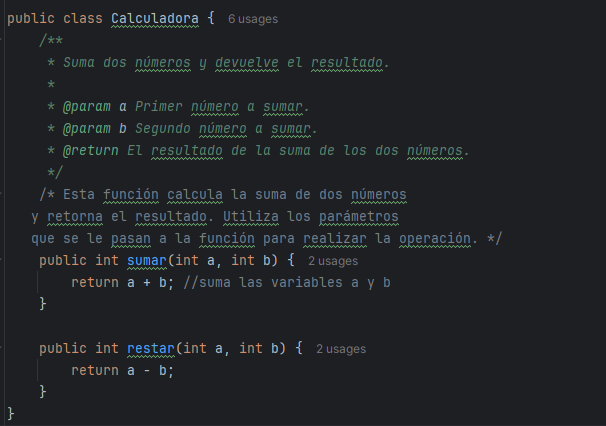
Mejora la comprensión del código.

Facilita la colaboración en equipo.

Ayuda a la localización de errores.

Permite una mejor toma de decisiones técnicas.

Favorece la escalabilidad y mantenimiento del proyecto.



Utilizar herramientas del entorno de desarrollo para documentar las clases.

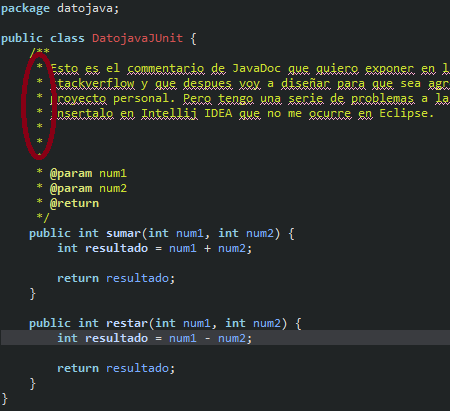
Utilizar herramientas del entorno de desarrollo para documentar las clases significa aprovechar las funciones que ofrecen los entornos de desarrollo integrados (IDE), como IntelliJ IDEA, Eclipse o Visual Studio, para dejar claro qué hace cada clase en un programa, cuál es su propósito, qué métodos contiene y cómo se relaciona con otras partes del sistema.

La documentación de clases ayuda a que otros programadores (y uno mismo en el futuro) puedan entender rápidamente **qué hace una clase** sin tener que leer todo el código en detalle. También facilita tareas como mantenimiento, pruebas, depuración y ampliación del software.

1. **Comentarios estructurados (como Javadoc en Java)**:  
   Son bloques de comentarios que se colocan justo encima de una clase o sus métodos. Permiten generar documentación automática legible desde el navegador o integrada en el IDE.
2. **Etiquetas de documentación**:  
   En el caso de Javadoc, se pueden usar etiquetas como:
   1. @author
   2. @version
   3. @see
   4. @param (para métodos)
   5. @return (para métodos)
3. **Diagramas y herramientas visuales del IDE**:  
   Algunos IDE permiten crear diagramas UML directamente desde el código, o mostrar estructuras jerárquicas de clases para entender mejor su organización.
4. **Comentarios en el editor**:  
   Muchos editores muestran la documentación de la clase automáticamente al colocar el cursor sobre su nombre o al usar atajos de teclado, si se ha documentado correctamente.

En una clase lo recomendado a documentar es:

* Su propósito general: qué representa o qué problema resuelve.
* Qué atributos tiene y qué representan.
* Qué métodos incluye y qué hacen.
* Si hereda de otra clase o implementa una interfaz, y por qué.
* Notas especiales o advertencias, si hay limitaciones o particularidades.



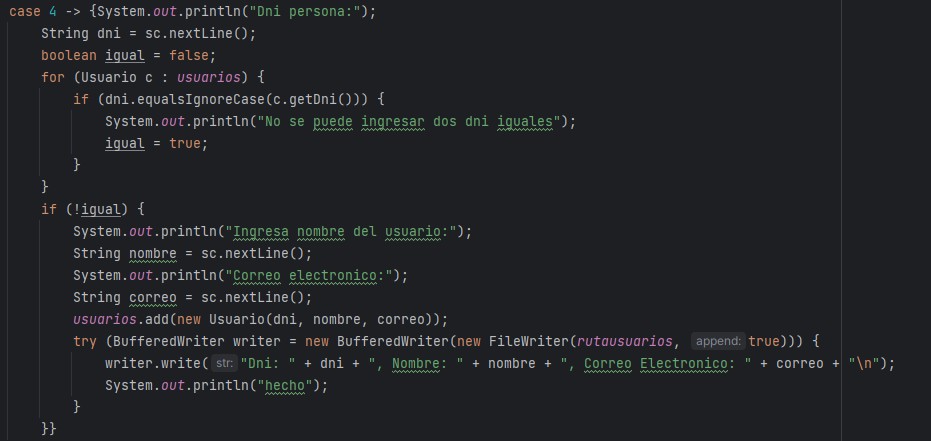
Refactorizacion de codigo

La refactorización en programación consiste en reorganizar y optimizar el código existente sin alterar su funcionalidad externa, con el objetivo de mejorar su legibilidad, mantenibilidad y estructura interna, eliminando redundancias, simplificando lógica compleja y aplicando buenas prácticas como el principio DRY (Don't Repeat Yourself) o SOLID. Este proceso se realiza para facilitar futuras modificaciones, reducir la deuda técnica y preparar el código para escalar, ya sea antes de añadir nuevas features, después de corregir bugs o durante revisiones de código, usando técnicas como extraer métodos, renombrar variables, reemplazar condicionales con polimorfismo o introducir patrones de diseño, todo ello manteniendo los mismos resultados pero con un código más limpio y eficiente.

1. Extraer método

Antes

Consiste en crear una función a partir de un fragmento de codigo para asi poder reutilizarlo

Después



2. Método en linea

Cuando un código es muy simple, es mejor poner el código directamente en vez de un método para así poder entenderlo mejor.

Antes



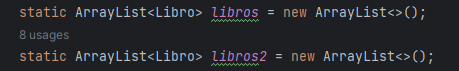
Después



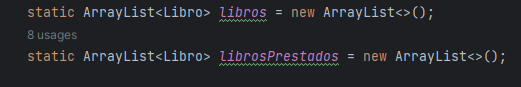
3. Renombrar

Esto consiste en cambiar nombres de funciones o variables de forma que sea mas claro y comprensible

Antes



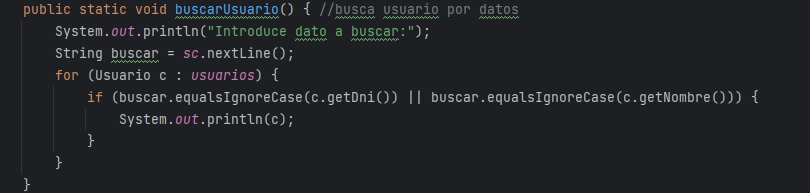
Después



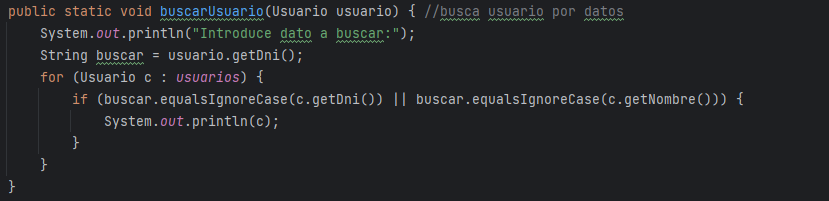
4. Objeto como parámetro

Se utiliza para disminuir las llamadas en los métodos, reduciendo las variables dentro de cada uno de estos.

Antes



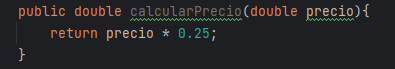
Después



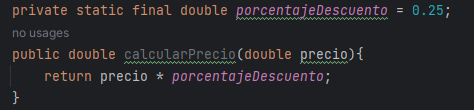
5. Numero como constantes

Consiste en definir todos los números que en un futuro puedan ser alterados, para así poder entender de donde procede dicho valor.

Antes



Después



Bibliografía

Casos de prueba: [https://es.linkedin.com/pulse/qué-es-un-caso-de-prueba- servicios-de-informatica-profesion-gyyzf](https://es.linkedin.com/pulse/qué-es-un-caso-de-prueba-servicios-de-informatica-profesion-gyyzf)

Herramientas de depuración: <https://www.ibm.com/es-es/topics/debugging>

Herramientas de prueba: <https://www.ibm.com/es-es/topics/software-testing>

Documentación del plan de pruebas: <https://es.linkedin.com/advice/1/what-test-plan-why-important-skills-programming-ffboc?lang=es&lang=es>

Implementación de pruebas automáticas: <https://digital.ai/es/catalyst-blog/how-automated-software-testing-solutions-help-create-more-secure-mobile-apps/>

Anexo

