

PROMETEO

Unidad 5: Pruebas y depuración de software

Entornos de desarrollo

Tecnico superior Administración y sistemas informáticos



Sesión 16: Introducción a las pruebas, verificación vs. validación

Probar software no es un trámite ni una fase final: es una disciplina completa que asegura la **calidad del producto** y la **confianza del usuario**. En desarrollo, el testing tiene un propósito claro: **detectar errores antes de que lleguen al cliente final**, minimizando riesgos, costes y pérdidas de reputación.

Cuando hablamos de **pruebas de software**, nos referimos a un conjunto de actividades planificadas que permiten comprobar si un sistema cumple con lo que se esperaba de él. Sin embargo, probar no es "buscar errores por error"; es un proceso sistemático que **valida la funcionalidad, verifica la implementación y garantiza la satisfacción del usuario**.

La diferencia entre verificación y validación

Aunque a menudo se usan como sinónimos, **verificación** y **validación** representan dos dimensiones complementarias del control de calidad:

Verificación responde a la pregunta "*¿Estamos construyendo el producto correctamente?*". Se centra en que el software cumpla las **especificaciones técnicas y los requisitos del diseño**. Aquí se revisa el código, la estructura y los componentes internos. En otras palabras, busca confirmar que el producto **sigue el plano original**.

Validación, en cambio, responde a "*¿Estamos construyendo el producto correcto?*". Su enfoque es externo: evalúa si el producto final **resuelve las necesidades reales del usuario o del negocio**. A veces, un software puede estar técnicamente bien construido (verificado) pero no cumplir con las expectativas del cliente (no validado).

Ambos procesos deben convivir. La verificación se realiza durante el desarrollo; la validación, cuando el sistema ya puede ser evaluado por usuarios o testers externos. Sin una buena verificación, no hay base sólida; sin validación, no hay producto útil.

Las etapas del testing

El proceso de pruebas se organiza habitualmente en **cuatro niveles**, cada uno con objetivos específicos:

- **Pruebas unitarias:** se centran en pequeñas partes del código (funciones o módulos). Buscan asegurar que cada componente aislado funciona correctamente.
- **Pruebas de integración:** evalúan cómo interactúan entre sí los distintos módulos. A menudo, los errores surgen no en el código individual, sino en la comunicación entre partes.
- **Pruebas de sistema:** verifican el comportamiento del sistema completo en un entorno similar al real. Se comprueba que todas las funcionalidades trabajen de forma coherente.
- **Pruebas de aceptación:** se realizan con el cliente o usuario final. El objetivo es validar que el sistema cumple los requisitos funcionales y las expectativas del negocio.

La cultura de la calidad

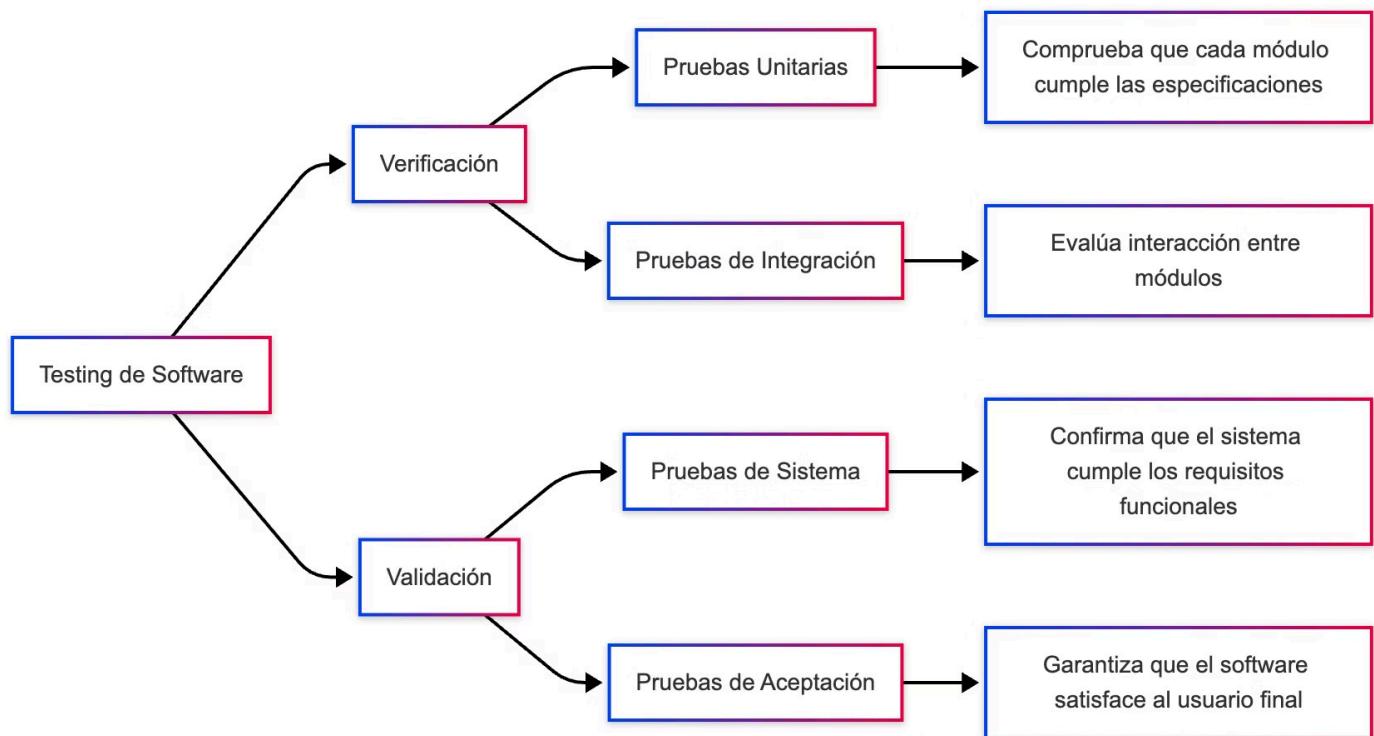
Probar no es solo tarea del equipo de QA (Quality Assurance). Cada miembro del proyecto —analistas, desarrolladores, diseñadores o gestores— tiene la responsabilidad de garantizar la calidad en su ámbito. Un código bien probado desde el inicio **reduce los costes de corrección**, acelera la entrega y aumenta la satisfacción del cliente.

En resumen:

- **Verificación** garantiza la corrección técnica.
- **Validación** garantiza la adecuación funcional.
- Ambas, juntas, garantizan la calidad total.



Esquema visual



ⓘ Descripción del esquema:

- En la parte izquierda, la **verificación** agrupa las pruebas técnicas (unitarias e integración).
- En la parte derecha, la **validación** se enfoca en las pruebas funcionales y de usuario (sistema y aceptación).
- Ambas fluyen dentro de un mismo proceso de testing, formando un **ciclo de control de calidad completo**, donde cada nivel se apoya en el anterior para garantizar fiabilidad y satisfacción.

Caso de estudio

BBVA y el control de calidad en aplicaciones financieras

Contexto

El sector bancario opera bajo una premisa clave: **la precisión no es negociable**. Un error de cálculo en una operación puede afectar a miles de clientes y dañar la confianza en la entidad. En 2018, BBVA detectó un fallo en la verificación del módulo de cálculo de intereses en su aplicación móvil de cuentas corrientes. Algunos saldos mostraban redondeos incorrectos debido a una inconsistencia en la función de cálculo.

Estrategia

El equipo técnico revisó todo el ciclo de pruebas e introdujo un nuevo proceso basado en **automatización y verificación continua**:

01	02	03
Pruebas unitarias automatizadas Se implementaron con JUnit (para Java) y PyTest (para microservicios Python).	Política de "no merge sin test superado" Ningún fragmento de código podía integrarse al repositorio principal sin pasar las pruebas automáticas.	Pruebas de aceptación de usuario (UAT) En colaboración con empleados de distintas oficinas, simulando transacciones reales.

Resultado

En menos de seis meses, el número de errores detectados por los usuarios se redujo un **80%**. Además, la automatización de la verificación permitió detectar fallos críticos en cuestión de minutos en lugar de días. El sistema de pruebas diarias ("daily verification build") se convirtió en una práctica estándar en todos los proyectos tecnológicos del banco.

Lección clave: el testing eficaz no es un coste adicional, sino una **inversión en confianza**. BBVA logró mayor estabilidad técnica, una reducción de incidencias y una mejora visible en la satisfacción del cliente.

Herramientas y consejos

El testing profesional combina **metodología, herramientas y disciplina**. A continuación, encontrarás las más utilizadas en el sector y buenas prácticas que debes incorporar desde el primer día:

Aplica pruebas desde el inicio del desarrollo

No esperes a que el software esté "listo".

Implementa **pruebas unitarias**

automáticas desde el primer sprint. Así detectas errores antes de que crezcan. En entornos ágiles, se aplica el principio "**Shift Left Testing**", que consiste en mover las pruebas hacia las fases iniciales del proyecto.

Usa frameworks de testing según el lenguaje

- **JUnit** para Java: el estándar en pruebas unitarias.
- **PyTest** para Python: sintaxis sencilla y muy compatible con CI/CD.
- **NUnit** para .NET: ideal en entornos Microsoft.
- **Jest** o **Mocha** para JavaScript y React.

Estos frameworks permiten automatizar, documentar y ejecutar cientos de casos de prueba en segundos.

Integra tus pruebas en un pipeline de CI/CD

Con herramientas como **Jenkins**, **GitHub Actions** o **GitLab CI**, puedes automatizar el testing cada vez que alguien sube código. Esto evita que se incorporen errores al repositorio principal y genera reportes automáticos de resultados.

Documenta los resultados y los casos

Cada prueba debe estar documentada con: objetivo, condiciones, pasos, resultado esperado y real. Utiliza plantillas estándar (por ejemplo, **TestRail** o **Xray** en Jira). Esta trazabilidad es vital para auditorías y certificaciones ISO.

Realiza pruebas de regresión tras cada cambio

Cada modificación de código, por pequeña que sea, puede romper otra parte del sistema. Automatiza las **pruebas de regresión** para asegurarte de que las funciones que ya estaban correctas siguen siéndolo.

No olvides las pruebas de seguridad

En sectores sensibles (finanzas, sanidad, educación), la validación también debe contemplar aspectos de seguridad. Herramientas como **OWASP ZAP** o **Burp Suite** ayudan a detectar vulnerabilidades antes del despliegue.

Mitos y realidades

 **Mito:** "Las pruebas son responsabilidad del equipo de QA."

→ **FALSO.** La calidad no es exclusiva del tester. Cada desarrollador debe probar su propio código antes de integrarlo. El QA verifica la calidad global, pero el control comienza con quien escribe el código. En metodologías ágiles, los equipos multidisciplinarios asumen la **responsabilidad compartida del testing.**

 **Mito:** "Si el software pasa todas las pruebas, está libre de errores."

→ **FALSO.** Las pruebas reducen el riesgo, pero no lo eliminan al 100%. Siempre pueden aparecer escenarios no contemplados o errores en condiciones extremas. El objetivo del testing no es garantizar perfección, sino **minimizar el riesgo y aumentar la confianza** en la entrega.

Resumen final

- **Testing = calidad preventiva.** Se detectan fallos antes de llegar al usuario.
- **Verificación:** comprueba que el software se construye correctamente.
- **Validación:** confirma que se construye el software correcto.
- **Tipos de pruebas:** unitarias, integración, sistema y aceptación.
- **Responsabilidad compartida:** todos los desarrolladores deben probar su código.



Sesión 17: Técnicas de prueba: caja negra y caja blanca

En el ámbito del testing, no todas las pruebas se ejecutan del mismo modo ni persiguen el mismo objetivo. El enfoque que adoptemos dependerá de **qué queremos comprobar**: el funcionamiento visible del sistema o la lógica interna del código. De esa distinción nacen las dos grandes técnicas universales de prueba de software: la **caja negra** y la **caja blanca**.

Ambas son esenciales. La caja negra simula la perspectiva del **usuario o cliente**, que interactúa con el sistema sin conocer su interior. La caja blanca adopta la visión del **desarrollador o ingeniero**, que analiza cómo fluye la información dentro del programa.

Caja negra: ¿Qué hace el sistema?

La **prueba de caja negra** evalúa el software desde el exterior. El tester no necesita ver ni conocer el código fuente; solo importa que, ante determinadas entradas, el sistema produzca las salidas correctas.

Su objetivo principal es **verificar la funcionalidad y el comportamiento** del sistema según los requisitos definidos. Por eso se utiliza ampliamente en fases de validación y control de calidad (QA).

Ejemplo: Si tienes una calculadora digital y escribes "2 + 2", la prueba de caja negra solo comprobará que el resultado mostrado sea "4". No importa cómo el código realiza la suma, sino que la función cumpla con lo prometido.

Entre las técnicas más utilizadas dentro de la caja negra se encuentran:

- **Partición de equivalencia:** divide el conjunto de posibles entradas en grupos o clases que se comportan igual. Si una entrada de la clase funciona, se asume que las demás también.
- **Análisis de valores límite:** prueba los extremos del rango de valores válidos y los límites inmediatos fuera de ese rango, ya que ahí suelen concentrarse los errores.
- **Pruebas de decisión y casos de uso:** validan que las condiciones lógicas y los escenarios funcionales se cumplan según lo esperado.

Caja blanca: ¿Cómo lo hace internamente?

La **prueba de caja blanca** (también conocida como estructural o de cristal) explora el interior del código. Aquí el objetivo no es solo verificar que el resultado sea correcto, sino **entender cómo se alcanza**. El tester debe conocer la estructura, los algoritmos y las rutas de ejecución.

Este enfoque permite identificar errores como:

- Bloques de código no ejecutados.
- Condiciones lógicas que nunca se cumplen.
- Bucles infinitos o mal definidos.
- Falta de cobertura en rutas críticas.

Entre sus técnicas más comunes destacan:

- **Cobertura de sentencias:** verificar que todas las líneas del código se ejecutan al menos una vez.
- **Cobertura de decisiones:** comprobar que cada condición booleana se evalúe tanto en verdadero como en falso.
- **Cobertura de caminos:** garantizar que todas las rutas posibles del flujo de ejecución se han probado.

En la práctica, mientras la caja negra busca responder a la pregunta "**¿Hace lo que debe hacer?**", la caja blanca responde "**¿Cómo lo hace internamente?**".

Ambas técnicas, aunque diferentes, son **complementarias y necesarias**. La caja negra detecta fallos funcionales visibles; la caja blanca revela defectos ocultos en la lógica interna. Solo al combinar ambas podemos asegurar una cobertura de pruebas completa y un producto confiable.



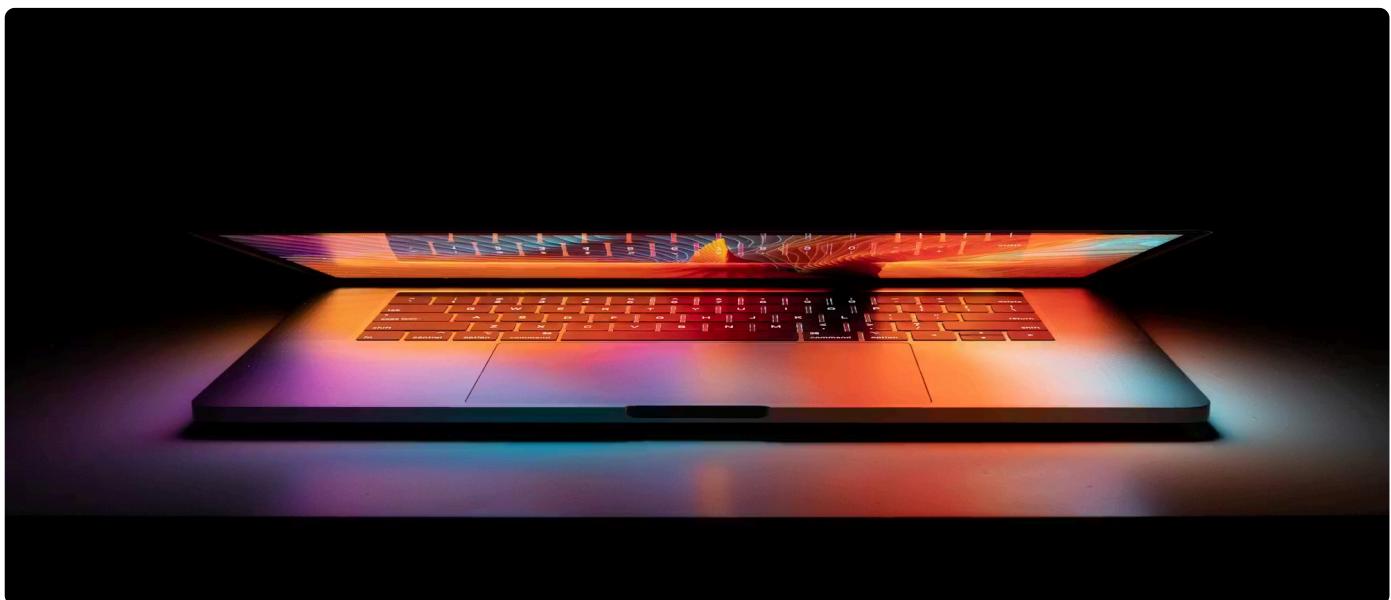
Esquema visual

El siguiente esquema resume visualmente la relación, el enfoque y el uso de ambas técnicas de testing.

Caja Negra	Caja Blanca
Enfoque externo	Enfoque interno
<ul style="list-style-type: none">• Evalúa entradas y salidas• Sin acceso al código• Se centra en la funcionalidad	<ul style="list-style-type: none">• Evalúa la lógica interna• Con acceso al código• Se centra en la estructura
Técnicas: Equivalencia, Valores límite, Casos de uso	Técnicas: Cobertura de sentencias, decisiones y caminos

ⓘ Descripción del esquema:

- La **caja negra** parte del comportamiento externo: el tester introduce datos, observa las respuestas y compara con el resultado esperado.
- La **caja blanca** examina el interior del sistema: revisa funciones, condiciones, bucles y rutas lógicas.
- Ambas fluyen desde un mismo nodo ("Técnicas de Prueba") porque se complementan: la primera valida la **correctitud funcional**, la segunda la **correctitud estructural**.



Caso de estudio

Ubisoft y la fiabilidad del motor gráfico

Contexto

En la industria del videojuego, la calidad del software es crítica. Un pequeño error en la lógica del motor puede provocar fallos gráficos, cierres inesperados o pérdida de datos del jugador. Durante el desarrollo de **Assassin's Creed Valhalla (Ubisoft, 2020)**, el equipo de control de calidad se enfrentó a un problema complejo: los jugadores experimentaban colisiones erróneas en escenarios 3D, donde los personajes "atravesaban" objetos sólidos.

Estrategia

El equipo de testing de Ubisoft combinó técnicas de **caja negra** y **caja blanca** para abordar el problema:

Caja negra

Los testers funcionales recrearon escenarios de juego con distintas configuraciones de personajes, entornos y velocidades para detectar cuándo y cómo se reproducían los errores. Analizaron el comportamiento visible sin entrar en el código.

Caja blanca

Paralelamente, los ingenieros de software revisaron el motor de físicas y descubrieron que el error se debía a una condición lógica en la función de detección de colisiones: un bloque "if" no cubría todos los posibles ángulos de impacto.

Resultado

Tras aplicar ambas técnicas, el error fue eliminado y la estabilidad del juego mejoró un **93%** en las métricas de fallos registrados en QA. El caso demostró que la **colaboración entre testers funcionales y desarrolladores técnicos** es clave: la caja negra detecta *qué* falla; la caja blanca explica *por qué*.

Conclusión del caso

Gracias a esta integración, Ubisoft consolidó un proceso interno llamado "**Dual QA Framework**", donde cada módulo del juego pasa primero por pruebas funcionales (caja negra) y luego por revisiones estructurales (caja blanca). Este modelo se ha convertido en un estándar dentro de su pipeline global de desarrollo.

Herramientas y consejos

Dominar ambas técnicas implica combinar la mentalidad analítica del programador con la mirada crítica del usuario. A continuación, te comparto herramientas y prácticas aplicables a cada enfoque:

Herramientas para Pruebas de Caja Negra

1. **Selenium** – ideal para automatizar pruebas funcionales de interfaces web. Permite simular interacciones reales del usuario.
2. **Postman** – perfecto para pruebas de APIs: envía peticiones HTTP y valida las respuestas esperadas sin ver el código backend.
3. **TestLink o Zephyr** – herramientas para documentar y gestionar casos de prueba, registrando entradas, salidas esperadas y resultados.

Herramientas para Pruebas de Caja Blanca

1. **JUnit / NUnit / PyTest** – frameworks para crear y ejecutar pruebas unitarias directamente sobre el código.
2. **SonarQube** – analiza la calidad del código y genera reportes de cobertura de pruebas, complejidad ciclomática y duplicidades.
3. **JaCoCo o Coverage.py** – miden la cobertura de líneas y rutas ejecutadas en cada test, un indicador esencial en caja blanca.

Consejos Profesionales

- Combina ambos enfoques desde el diseño del sistema

No esperes a tener el producto final para aplicar la caja negra; y no limites la caja blanca a los desarrolladores.

- Integra pruebas automáticas en tu pipeline de CI/CD

Configura Jenkins o GitHub Actions para ejecutar tests tras cada commit. Así garantizas que cada versión esté verificada y validada.

- Documenta los resultados

Usa una matriz de trazabilidad entre requisitos, casos de prueba y defectos encontrados. Esto mejora la comunicación entre desarrollo, QA y negocio.

- Analiza métricas de cobertura

Un alto porcentaje no garantiza ausencia de errores, pero sí demuestra disciplina y control del proceso.

- Prioriza los casos críticos

No todos los escenarios necesitan la misma profundidad. Identifica los módulos de mayor riesgo (por impacto o frecuencia de uso) y enfoca ahí los esfuerzos.

Mitos y realidades

 **Mito:** "La caja blanca sustituye a la caja negra."

→ **FALSO.** Ninguna puede reemplazar a la otra. La caja blanca se enfoca en el código, mientras la caja negra observa el comportamiento desde fuera. Un software puede tener una lógica impecable internamente y, aun así, no cumplir con las expectativas del usuario si no se valida externamente.

 **Mito:** "La caja negra no requiere conocimientos técnicos."

→ **FALSO.** Aunque el tester no necesita leer código, sí debe comprender la lógica funcional, los flujos de negocio y los escenarios de uso. La efectividad de las pruebas depende de la capacidad de diseñar casos realistas y relevantes.

Realidad:

El éxito del testing proviene del equilibrio entre **visión técnica** (caja blanca) y **visión funcional** (caja negra). En los equipos modernos de QA, ambos perfiles trabajan en conjunto desde las fases tempranas del desarrollo.

Resumen final

- **Caja negra:** evalúa las entradas y salidas del sistema sin ver el código. Se centra en la funcionalidad y la experiencia del usuario.
- **Caja blanca:** analiza la lógica interna, rutas y condiciones del código. Evalúa la estructura y la eficiencia del programa.
- **Ambas son complementarias:** una asegura el cumplimiento funcional (validación), la otra garantiza la corrección técnica (verificación).
- **Herramientas clave:** Selenium, Postman, JUnit, SonarQube.
- **Objetivo final:** aumentar la cobertura de pruebas y reducir el riesgo de fallos antes de la entrega.

Sesión 18: Pruebas de integración, regresión y no funcionales

Una vez que los componentes de un software funcionan de forma aislada, llega el verdadero reto: **verificar cómo se comportan juntos**. Un programa rara vez actúa como una pieza única; está formado por módulos, servicios y dependencias que deben comunicarse correctamente. En ese punto entran las **pruebas de integración**, las **pruebas de regresión** y las **pruebas no funcionales**, tres pilares esenciales para garantizar la calidad real de un sistema.

Estas pruebas ya no buscan saber si el código *compila o ejecuta correctamente*, sino **cómo interactúan las partes y cómo responde el sistema bajo condiciones reales de uso**.

Pruebas de integración: asegurando que las piezas encajan

Las **pruebas de integración** tienen un objetivo claro: verificar que los distintos módulos del software se comunican correctamente entre sí.

Un ejemplo clásico sería un sistema de e-commerce: la funcionalidad de "añadir al carrito" debe conectar el módulo del catálogo con el de inventario, el de usuario y el de pagos. Aunque cada uno funcione individualmente, un error en su interacción puede provocar fallos graves.

Existen distintos enfoques para realizarlas:

- **Integración ascendente (bottom-up)**: se prueban primero los módulos más pequeños o de bajo nivel (funciones o servicios) y luego se combinan progresivamente hacia módulos superiores.
- **Integración descendente (top-down)**: se empieza por los módulos de alto nivel (interfaces, controladores) y se simulan los inferiores con *stubs* (funciones temporales).
- **Integración continua**: propia de entornos DevOps, donde cada cambio en el código desencadena automáticamente un conjunto de pruebas que validan la integración global del sistema.

El propósito es claro: **no basta con que cada parte funcione; deben hacerlo en conjunto y sin interferencias**.

Relación entre los tres tipos de pruebas

Podemos entender la relación así:

- **Integración**: aseguran la comunicación entre módulos.
- **Regresión**: verifican que el sistema no se rompa tras cambios.
- **No funcionales**: confirman que el producto cumple estándares de calidad global.

Juntas, permiten entregar software estable, escalable y confiable.

Pruebas de regresión: asegurar que lo nuevo no rompe lo anterior

Cada vez que se modifica o amplía un sistema, existe el riesgo de que un cambio altere funcionalidades que antes funcionaban correctamente. Las **pruebas de regresión** buscan precisamente prevenir eso.

Se trata de un conjunto de casos de prueba que se ejecutan cada vez que hay una nueva versión o actualización. Su misión es confirmar que el software **mantiene la estabilidad y coherencia** tras los cambios.

Por ejemplo: si un desarrollador modifica la función de cálculo de descuentos en una tienda online, las pruebas de regresión revisarán que el cálculo de impuestos, el envío y el total final sigan funcionando correctamente.

Las empresas modernas automatizan este tipo de pruebas para reducir tiempos y evitar errores humanos. En entornos **CI/CD (Integración y Entrega Continua)**, se ejecutan automáticamente después de cada compilación o *merge* de código.

Pruebas no funcionales: rendimiento, seguridad y usabilidad

Hasta aquí, las pruebas se enfocaban en **qué hace el sistema**. Pero igual de importante es **cómo lo hace**.

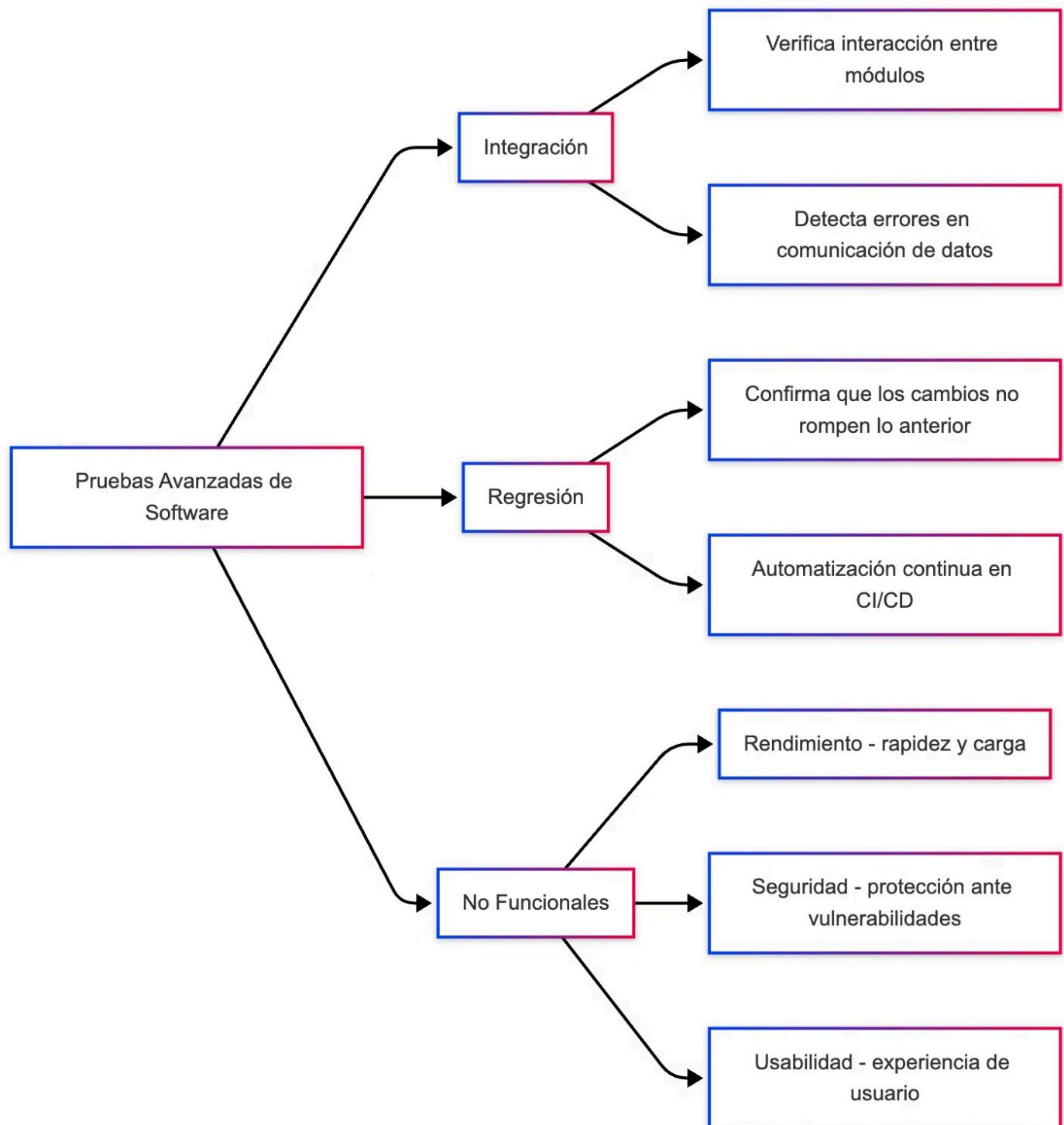
Las **pruebas no funcionales** miden los atributos de calidad que no están directamente ligados a la funcionalidad, sino al **rendimiento global, la seguridad, la compatibilidad y la experiencia del usuario**.

Entre las más habituales destacan:

1. **Pruebas de rendimiento (performance testing):** Evalúan la velocidad, capacidad de respuesta y estabilidad del sistema bajo distintas cargas.
 - *Stress test*: mide hasta qué punto el sistema soporta niveles extremos.
 - *Load test*: analiza cómo responde ante un número creciente de usuarios simultáneos.
 - *Soak test*: examina el comportamiento tras largas horas de uso continuo.
2. **Pruebas de seguridad:** Detectan vulnerabilidades que puedan ser explotadas (inyección SQL, XSS, errores de autenticación). En muchos casos se emplean herramientas automatizadas como **OWASP ZAP** o **Burp Suite**.
3. **Pruebas de compatibilidad y usabilidad:** Validan que el software funcione correctamente en diferentes dispositivos, navegadores o sistemas operativos. También miden la facilidad de uso desde la perspectiva del usuario final.

En conjunto, estas pruebas garantizan que el software no solo "funcione", sino que lo haga **de forma eficiente, segura y cómoda**.

Esquema visual



ⓘ Descripción del esquema:

- Las **pruebas de integración** comprueban cómo se combinan los módulos.
- Las **de regresión** se centran en la estabilidad del software tras actualizaciones.
- Las **no funcionales** evalúan la calidad global desde la perspectiva del rendimiento, la seguridad y la experiencia del usuario. Juntas, conforman la capa final del proceso de testing antes del despliegue en producción.

Caso de estudio

Amazon y la automatización de pruebas en su plataforma de e-commerce

Contexto

Amazon gestiona una de las infraestructuras más complejas del mundo digital: miles de microservicios conectados (catálogo, inventario, logística, pagos, recomendaciones, atención al cliente...). Cualquier error de integración podría generar pérdidas millonarias.

En 2022, tras una actualización en el módulo de recomendaciones, Amazon detectó un aumento de errores 500 en el backend. El fallo no estaba en el nuevo código, sino en la **integración** con el servicio de inventario, que no reconocía ciertos parámetros JSON.

Estrategia

El equipo técnico aplicó una combinación de pruebas de **integración, regresión y no funcionales**:

• Integración

Se implementó un sistema de pruebas automáticas que validaba las APIs internas mediante **Postman y Newman**, asegurando que cada servicio devolviera respuestas coherentes con los contratos definidos.

• Regresión

Se incorporó un pipeline de pruebas en **AWS CodeBuild**, de modo que cada actualización en el repositorio ejecutaba cientos de test automatizados para garantizar que ninguna funcionalidad previa se rompiera.

• No funcionales

Finalmente, se evaluó el rendimiento con **JMeter** bajo carga simulada de más de 10.000 solicitudes por minuto, comprobando que los tiempos de respuesta no superaran los 200 ms promedio.

Resultado

Gracias a este enfoque integral:

- Se redujeron los fallos en producción un **92%**.
- El tiempo de despliegue se acortó de 12 horas a 90 minutos.
- Los equipos de QA pudieron centrarse en pruebas exploratorias en lugar de tareas repetitivas.

Conclusión: Amazon demostró que la combinación de pruebas automáticas y no funcionales no solo previene errores, sino que **aumenta la eficiencia del desarrollo continuo y la confianza del cliente**.

Herramientas y consejos

El testing avanzado requiere herramientas específicas para cada tipo de prueba. A continuación, se destacan las más utilizadas y cómo aplicarlas en entornos reales:

1. Automatiza las pruebas de regresión

- **JUnit (Java)** o **PyTest (Python)** son ideales para ejecutar automáticamente tests cada vez que se realiza un *commit*.
- **Selenium WebDriver** permite automatizar pruebas funcionales sobre navegadores web.

Consejo: define un conjunto de *smoke tests* que se ejecuten tras cada build para detectar rápidamente fallos críticos.

2. Prueba la integración con APIs

- **Postman** y su versión en línea de comandos **Newman** son herramientas esenciales para probar endpoints, cabeceras, parámetros y respuestas de servicios web.

Configura colecciones con diferentes entornos (desarrollo, staging, producción) para detectar inconsistencias antes de los despliegues.

3. Evalúa el rendimiento del sistema

- **Apache JMeter** y **K6.io** permiten simular miles de usuarios concurrentes y medir métricas como el tiempo de respuesta, la tasa de errores o el throughput.

En proyectos web, monitoriza el rendimiento con **Lighthouse** (de Google Chrome) para detectar bloqueos o lentitud en el frontend.

4. Refuerza la seguridad

- **OWASP ZAP** es una herramienta gratuita que analiza vulnerabilidades web (inyección SQL, XSS, CSRF).

Automatiza pruebas de seguridad periódicas y genera reportes que prioricen riesgos según su criticidad.

5. Incorpora las pruebas en tu cultura DevOps

- Integra todos estos tests en el pipeline de **CI/CD** (GitHub Actions, Jenkins o GitLab).

Cada despliegue debe pasar automáticamente por una secuencia: *unitarias → integración → regresión → no funcionales*.

Esto no solo mejora la calidad, sino que permite **entregas continuas sin perder estabilidad**.

Mitos y realidades

 **Mito:** "Las pruebas de integración solo son necesarias en sistemas grandes."

→ **FALSO.** Incluso aplicaciones pequeñas (como una app de notas o una calculadora con base de datos) pueden fallar por errores en la interacción entre módulos. Toda comunicación entre componentes merece ser verificada.

 **Mito:** "Las pruebas no funcionales son secundarias."

→ **FALSO.** De poco sirve que una app "funcione" si tarda 10 segundos en cargar o se cae con 100 usuarios conectados. El rendimiento y la seguridad son tan importantes como la funcionalidad.

Realidad:

Las pruebas avanzadas (integración, regresión y no funcionales) son la **frontera entre un software que funciona y un software profesional y escalable**.

Resumen final

- **Pruebas de integración:** verifican que los módulos colaboran correctamente.
- **Pruebas de regresión:** garantizan que los cambios no rompan lo que ya funcionaba.
- **Pruebas no funcionales:** evalúan rendimiento, seguridad y usabilidad.
- **Herramientas clave:** JUnit, Selenium, Postman, JMeter, OWASP ZAP.
- **Clave profesional:** automatiza y mide continuamente; la calidad no se prueba al final, se construye desde el inicio.

Sesión 19: Métricas de calidad: complejidad ciclomática y CRAP

En desarrollo de software, "que funcione" no es suficiente. La verdadera calidad no se mide solo por la ausencia de errores, sino por **lo fácil que resulta mantener, probar y evolucionar el código a lo largo del tiempo**. Un sistema puede funcionar hoy, pero si su estructura es demasiado compleja o poco legible, se convertirá en un riesgo costoso a medio plazo.

Por eso surgen las **métricas de calidad del código**, indicadores objetivos que permiten evaluar la salud interna de un proyecto. Entre ellas, dos destacan por su relevancia profesional y su uso en auditorías técnicas: la **complejidad ciclomática** y el **CRAP score**.

Ambas métricas son herramientas cuantitativas que permiten responder a preguntas críticas:

- ¿Qué tan difícil será mantener o probar este código?
- ¿Qué módulos presentan mayor riesgo de errores o regresiones?
- ¿Dónde conviene invertir esfuerzos de refactorización?

Complejidad ciclomática: medir la estructura lógica del código

La **complejidad ciclomática (CC)** fue propuesta en 1976 por Thomas McCabe como una métrica que mide la cantidad de **rutas lógicas independientes** que existen dentro de un programa. En términos simples, indica **cuántos caminos diferentes puede seguir la ejecución del código**.

Cada condicional (if, else, while, for, switch, etc.) aumenta el número de posibles rutas y, por tanto, la complejidad. A mayor número de caminos, más difícil es garantizar que todos estén correctamente probados y que los cambios futuros no generen errores ocultos.

Fórmula de la complejidad ciclomática

La fórmula más común es:

$$\mathbf{CC = E - N + 2}$$

Donde:

- **E** = número de aristas (transiciones entre nodos del flujo de control).
- **N** = número de nodos (bloques de código o decisiones).

En la práctica, los IDEs modernos y herramientas como **SonarQube**, **CodeClimate** o **Visual Studio Metrics** calculan automáticamente esta métrica.

Interpretación de los valores

Complejidad Ciclomática	Nivel de Riesgo	Recomendación
1 - 10	Baja	Mantenible y fácil de probar
11 - 20	Moderada	Requiere atención y pruebas cuidadosas
21 - 30	Alta	Dificultad significativa para testear
> 30	Muy alta	Refactorización urgente recomendada

Un módulo con una complejidad de 8 es saludable; uno con 40 indica que probablemente contiene demasiadas decisiones, bucles o dependencias internas.

Ejemplo práctico

Imagina un método de validación de usuario con múltiples condicionales:

```
def validar_usuario(edad, suscrito, pais):
    if edad < 18:
        return "Rechazado"
    elif pais not in ["ES", "MX", "AR"]:
        return "No disponible"
    elif suscrito:
        return "Aprobado Premium"
    else:
        return "Aprobado"
```

Este pequeño bloque ya tiene **4 caminos lógicos diferentes**. Si aumentamos condiciones, excepciones o bucles, la complejidad crece rápidamente y con ella el riesgo de errores y dificultades de mantenimiento.

Métrica CRAP: evaluar el riesgo del código con base en complejidad y cobertura

Años después, se reconoció que medir solo la complejidad no era suficiente. Un código complejo puede ser perfectamente seguro **si está bien cubierto por pruebas automatizadas**.

Así nació el **CRAP score (Change Risk Anti-Patterns)**, una métrica moderna creada por Alberto Savoia y Bob Evans que combina la **complejidad ciclomática (CC)** con la **cobertura de pruebas (% coverage)**.

Fórmula

$$\text{CRAP} = \text{CC}^2 \times (1 - \text{Coverage})^3 + \text{CC}$$

Donde:

- **CC** = complejidad ciclomática del método.
- **Coverage** = cobertura de pruebas (en valor decimal; por ejemplo, 80% = 0.8).

Interpretación

CRAP Score	Evaluación	Riesgo
< 30	Bueno	Bajo riesgo
30 - 60	Regular	Moderado, se recomienda mejorar tests
> 60	Malo	Alto riesgo, código difícil de mantener

Un valor alto de CRAP indica una zona crítica del software: compleja, poco cubierta por pruebas y propensa a errores cuando se modifica.

Ejemplo

Supón un módulo con:

- Complejidad ciclomática = 12
- Cobertura de pruebas = 50% (0.5)

$$\text{CRAP} = 12^2 \times (1 - 0.5)^3 + 12 = 144 \times 0.125 + 12 = 30$$

→ El resultado (30) indica **riesgo moderado**, lo que sugiere aumentar la cobertura de pruebas o simplificar el código.

Por qué estas métricas importan

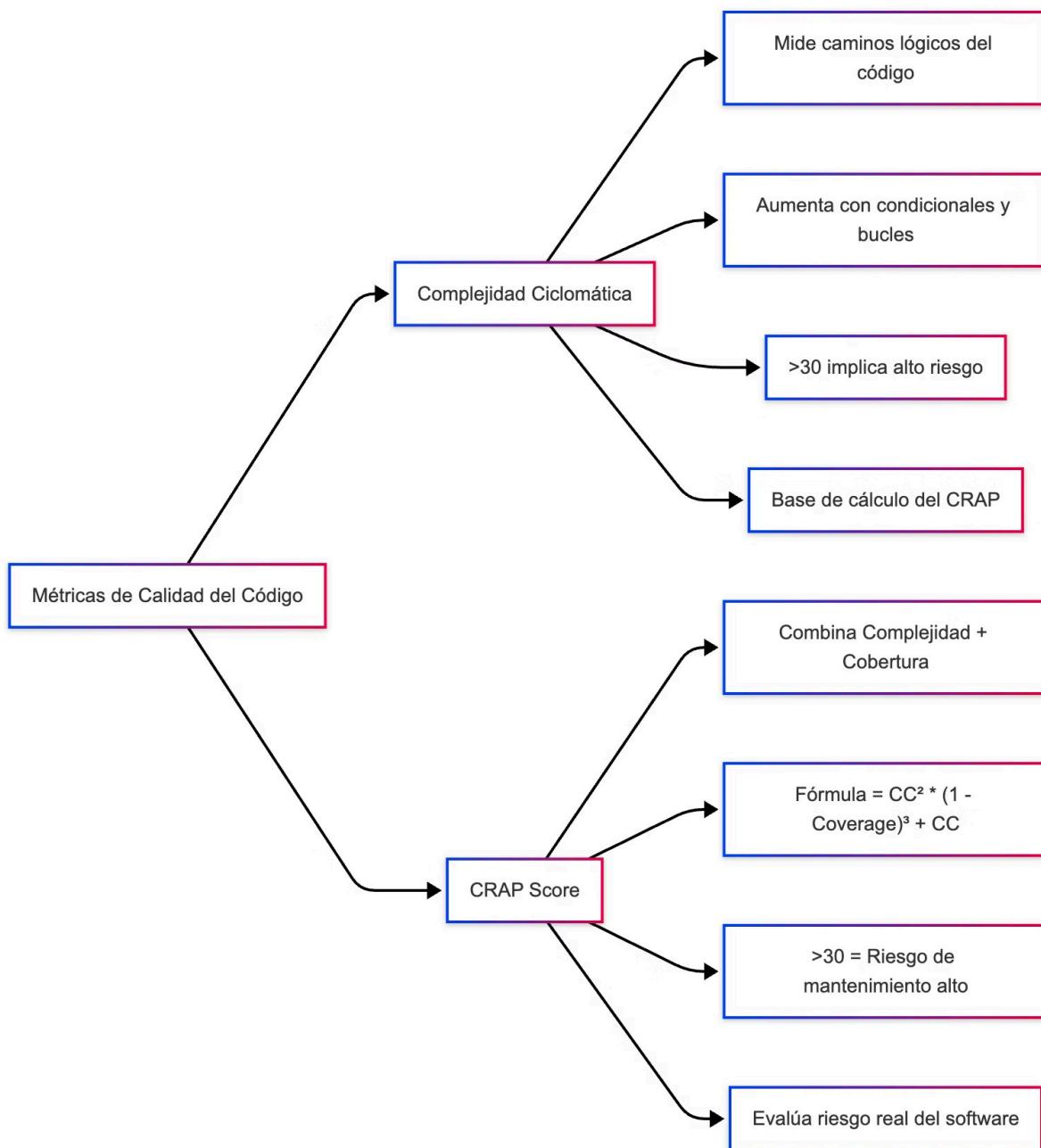
- **Permiten priorizar esfuerzos:** los equipos de QA y desarrollo pueden centrarse en las zonas más críticas.
- **Aumentan la mantenibilidad:** el código más limpio es más fácil de entender, probar y evolucionar.
- **Reducen costes futuros:** cada punto de complejidad elevado representa tiempo adicional en debugging y testeo.
- **Alinean desarrollo y calidad:** facilitan la comunicación técnica entre equipos y justifican decisiones de refactorización.

En definitiva, medir la complejidad no es una cuestión estética, sino económica y de sostenibilidad técnica.



Esquema visual

El siguiente esquema muestra la relación entre Complejidad Ciclomática y CRAP Score.



ⓘ Descripción del esquema:

- La **complejidad ciclomática** mide el número de caminos lógicos.
- El **CRAP score** utiliza esa información y la ajusta con el nivel de cobertura de pruebas.
- Si un código es complejo **y** poco probado, el riesgo se dispara.
- Ambos indicadores trabajan juntos para identificar zonas críticas del software que requieren refactorización o más pruebas.

Caso de estudio

Sanitas y la mejora de mantenibilidad en su sistema médico

Contexto

Sanitas, compañía española del sector sanitario, gestionaba un sistema de historia clínica con más de 2 millones de registros. El software funcionaba correctamente, pero los tiempos de despliegue y las incidencias de mantenimiento aumentaban cada trimestre.

Al realizar una auditoría técnica con **SonarQube**, se descubrió que varios módulos críticos tenían:

- Complejidad ciclomática superior a **45**.
- Cobertura de pruebas inferior al **15%**.

Estos módulos correspondían a cálculos de facturación médica y validación de tratamientos.

Estrategia

El equipo de ingeniería implantó un plan en tres fases:

01

Medición y visualización

Se integró **SonarQube** con el pipeline de Jenkins para generar reportes automáticos de complejidad y CRAP tras cada build.

02

Refactorización controlada

Los métodos con CRAP > 30 se dividieron en funciones más simples y se introdujeron pruebas unitarias y de integración.

03

Automatización de control de calidad

Se estableció una política de "no merge si CRAP > 30 o cobertura < 70%".

Resultado

En seis meses:

- La complejidad promedio del sistema bajó de **38 a 9**.
- La cobertura de pruebas subió de **18% a 82%**.
- Las incidencias postdespliegue se redujeron un **67%**.

Además, el tiempo de incorporación de nuevos desarrolladores disminuyó gracias a la mayor legibilidad del código.

Conclusión: medir y gestionar métricas de calidad transformó un sistema estable pero frágil en una plataforma sólida, escalable y sostenible.

Herramientas y consejos

1. Controla la complejidad desde el inicio

No esperes a tener código "legacy" para medir calidad. Configura tu entorno de desarrollo (por ejemplo, Visual Studio o IntelliJ) para mostrar la complejidad ciclomática por función.

2. Usa herramientas especializadas

- **SonarQube**: estándar industrial para auditorías de código, analiza CC, CRAP, duplicaciones y vulnerabilidades.
- **CodeClimate**: plataforma SaaS que evalúa calidad en proyectos alojados en GitHub o GitLab.
- **JaCoCo (Java)** o **Coverage.py (Python)**: miden cobertura de pruebas para alimentar el cálculo del CRAP.

3. Refactoriza funciones largas

Si un método supera 30 líneas o tiene múltiples "if-else", considera dividirlo en submétodos. Cada función debe tener **una sola responsabilidad** (principio SRP).

4. Aumenta la cobertura de tests

Automatiza pruebas unitarias con **JUnit**, **PyTest** o **NUnit**. Recuerda: un código con alta complejidad pero **95% de cobertura** puede seguir siendo fiable; uno con baja cobertura es un riesgo latente.

5. Revisa métricas en cada sprint

En entornos ágiles, incluye la revisión de métricas en las retrospectivas. Identificar módulos con alto CRAP debe ser parte del ciclo de mejora continua.

Mitos y realidades

 **Mito:** "Más líneas de código significa mejor software."

→ **FALSO.** La calidad no depende de la cantidad. El código excelente es breve, claro y cumple su propósito con la menor complejidad posible.

 **Mito:** "Las métricas solo importan a los auditores."

→ **FALSO.** Las métricas son guías para el propio desarrollador. Un profesional que conoce la complejidad de su código trabaja con criterio y anticipa problemas antes de que ocurran.

Realidad:

Un código de calidad no solo ejecuta correctamente, sino que es **predecible, mantenible y medible**. Las métricas son el espejo que muestra la salud real de un proyecto.

Resumen final

- **Complejidad ciclomática:** mide el número de caminos lógicos; ideal <10.
- **CRAP score:** combina complejidad y cobertura de pruebas; ideal <30.
- **Herramientas:** SonarQube, CodeClimate, JaCoCo, Coverage.py.
- **Claves profesionales:** mide, refactoriza y automatiza; un código limpio reduce costes y errores.
- **Principio esencial:** lo que no se mide, no se mejora.

Sesión 20: Depuración con IDEs y pruebas automatizadas con JUnit

Depurar es una de las habilidades más valiosas en la programación profesional. Significa **detenerse a comprender el error**, no solo corregirlo. En su definición formal, **depurar (debugging)** es el proceso sistemático de identificar, aislar y corregir defectos en el código fuente de un programa. Pero en la práctica, depurar es mucho más: es **aprender cómo piensa el código y cómo se comporta el sistema cuando algo no sale como esperábamos**.

Los entornos de desarrollo integrados (IDEs) modernos —como **IntelliJ IDEA, Eclipse, Visual Studio Code o NetBeans**— incluyen depuradores potentes que permiten ejecutar el programa **paso a paso**, inspeccionar variables, evaluar expresiones y observar cómo fluye la lógica interna del software. Gracias a estas herramientas, el programador no tiene que limitarse a "leer el código": puede **verlo vivir y reaccionar en tiempo real**.

¿Por qué es esencial la depuración?

En el ciclo de desarrollo, escribir código sin depurar es como construir una casa sin revisar los cimientos. Los errores (bugs) pueden permanecer ocultos durante semanas y emergir justo antes de la entrega. Depurar permite:

- Identificar el origen exacto de un fallo, no solo el síntoma.
- Comprender cómo se comporta el programa en ejecución.
- Validar hipótesis sobre la lógica del código.
- Evitar errores repetitivos mediante la observación del flujo interno.

La depuración, por tanto, no se limita a "arreglar bugs": es una **técnica de aprendizaje continuo sobre el sistema**.

Pruebas automatizadas: el complemento del debugging

Mientras la depuración analiza los errores en ejecución, las **pruebas automatizadas** previenen que vuelvan a aparecer. En vez de ejecutar manualmente el programa cada vez que cambias algo, puedes escribir **scripts de prueba** que lo hagan automáticamente, verificando si el código sigue comportándose como debería.

Las pruebas automatizadas ofrecen tres beneficios fundamentales:

1. **Confianza:** cada cambio en el código se valida de forma inmediata.
2. **Velocidad:** reducen el tiempo de verificación en grandes proyectos.
3. **Prevención:** evitan la reaparición de errores antiguos (regresiones).

JUnit: el estándar de testing en Java

JUnit es el framework de pruebas unitarias más utilizado en el ecosistema Java. Permite escribir métodos de prueba que comparan resultados esperados con resultados obtenidos de forma automatizada. Cada vez que se modifica el código, las pruebas se ejecutan automáticamente (desde el IDE o el pipeline de integración continua), asegurando que todo siga funcionando.

Estructura básica de una prueba con JUnit

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertEquals;  
  
public class CalculadoraTest {  
    @Test  
    void testSuma() {  
        Calculadora calc = new Calculadora();  
        int resultado = calc.sumar(2, 3);  
        assertEquals(5, resultado, "La suma debería ser 5");  
    }  
}
```

En este ejemplo:

- **@Test** indica que el método es una prueba.
- **assertEquals()** compara el valor esperado con el obtenido.
- Si ambos coinciden, la prueba pasa (verde). Si no, falla (rojo).

Esta estructura es la base de la **automatización del testing**. En proyectos reales, cientos o miles de tests se ejecutan automáticamente en cuestión de segundos, alertando a los equipos si algo falla tras un cambio.

Depuración + JUnit: un ciclo de calidad continuo

Ambas prácticas no compiten, se complementan:

- **Depuración:** detecta y corrige el error.
- **Pruebas automatizadas:** evitan que el error reaparezca.

En un flujo de trabajo profesional, cada vez que un desarrollador corrige un bug, se crea una **prueba unitaria** que reproduce ese fallo. De ese modo, si en el futuro alguien modifica el código y vuelve a provocar el error, el test lo detectará inmediatamente.

En los entornos **DevOps y CI/CD**, este enfoque garantiza una **validación continua** del software, reduciendo costes y tiempos de entrega.

Esquema visual

El siguiente esquema muestra el flujo de depuración y automatización con JUnit.

01

Error detectado en ejecución

Identificación de un comportamiento no esperado durante la ejecución del programa

02

Depuración con IDE

Uso de breakpoints, inspección de variables y ejecución paso a paso para localizar la causa

03

Identificar causa y corregir bug

Ánalisis del flujo lógico y modificación del código para eliminar el defecto

04

Crear prueba automatizada con JUnit

Escrivir un test que reproduzca el error y valide la corrección

05

Integrar prueba en pipeline CI/CD

Incorporación del test al sistema de integración continua para ejecución automática

06

Validación continua

Ejecución automática del test en cada cambio de código para prevenir regresiones

Descripción del esquema:

- El proceso comienza con un error identificado durante la ejecución.
- El depurador del IDE permite seguir el flujo paso a paso y localizar la causa raíz.
- Una vez corregido, se escribe una **prueba automatizada en JUnit** que garantiza que el problema no volverá a ocurrir.
- Finalmente, la prueba se integra en el ciclo de **integración continua (CI/CD)**, ejecutándose automáticamente en cada versión del software.



Caso de estudio

Contexto

Booking.com, una de las mayores plataformas de reservas online, sufrió en 2021 un error intermitente en su sistema de gestión de disponibilidad hotelera. En determinadas combinaciones de fechas, la web devolvía resultados nulos o reservas duplicadas. El problema aparecía solo bajo condiciones específicas, lo que lo hacía difícil de reproducir manualmente.

Estrategia

El equipo de desarrollo usó el **depurador de IntelliJ IDEA** para analizar la ejecución paso a paso en un entorno de pruebas. Colocaron **breakpoints** en el módulo que calculaba disponibilidad y descubrieron que una variable del tipo LocalDate se inicializaba incorrectamente cuando el rango de fechas cruzaba meses con distintos números de días.

Tras corregir el error, los ingenieros crearon **tests automatizados con JUnit** que simulaban escenarios límite (meses de 30 y 31 días, años bisiestos, etc.). Cada nuevo cambio en el módulo disparaba automáticamente las pruebas a través de **Jenkins**, asegurando que el bug no pudiera reaparecer.

Resultado

- El error se eliminó completamente en menos de 48 horas.
- Las nuevas pruebas redujeron en un 90% las incidencias similares durante los meses siguientes.
- El tiempo promedio de corrección de bugs pasó de 3 días a menos de 12 horas.

Conclusión: combinar depuración eficaz con pruebas automatizadas no solo resuelve problemas, sino que **construye una red de seguridad permanente** para el software.

Herramientas y consejos

1

1. Usa breakpoints estratégicos

Colocar puntos de interrupción en cada línea solo ralentiza el proceso. Ponlos en lugares clave: inicio de bucles, condiciones críticas o líneas donde cambian valores importantes.

2

2. Observa las variables en tiempo real

Los IDEs permiten abrir una **ventana de variables** donde puedes ver sus valores y cómo cambian con cada paso. Esto ayuda a descubrir comportamientos inesperados.

3

3. Integra JUnit con tu gestor de dependencias

En **Maven**, ejecuta las pruebas con:

```
mvn test
```

En **Gradle**:

```
./gradlew test
```

Ambos entornos permiten incluir los tests en la fase de build, lo que automatiza la validación del código antes del despliegue.

4

4. Conecta tus pruebas a CI/CD

Usa herramientas como **GitHub Actions**, **GitLab CI**, **Jenkins** o **CircleCI** para ejecutar automáticamente las pruebas cada vez que se realiza un *commit* o *pull request*. Esto asegura que el código defectuoso nunca llegue a producción.

5

5. Combina depuración con TDD

En la siguiente sesión veremos el enfoque **Test-Driven Development (TDD)**, donde las pruebas se escriben antes del código. Depurar y probar no deben ser pasos separados, sino parte del mismo flujo de aprendizaje y mejora.

Mitos y realidades

 **Mito:** "Depurar solo es necesario para errores graves."

→ **FALSO.** La depuración continua es una práctica preventiva. Permite detectar errores pequeños antes de que se conviertan en fallos críticos en producción.

 **Mito:** "Las pruebas automatizadas son una pérdida de tiempo."

→ **FALSO.** Escribir pruebas toma minutos; corregir errores en producción puede costar días, reputación y dinero. Los tests automatizados son una inversión en estabilidad y confianza.

Realidad:

Los entornos modernos combinan depuración, automatización y CI/CD para ofrecer **software de alta calidad en ciclos de desarrollo más cortos y controlados**.

Resumen final

- **Depuración:** ejecutar el código paso a paso para localizar errores.
- **Herramientas:** IntelliJ, Eclipse, VS Code, NetBeans.
- **JUnit:** framework estándar de pruebas automatizadas en Java.
- **Principales anotaciones:** @Test, assertEquals(), assertTrue().
- **CI/CD:** integración de pruebas automáticas en pipelines para detección continua.
- **Claves profesionales:** depura a menudo, automatiza tus tests y no confíes solo en la ejecución manual.



Sesión 21: TDD, ATDD y BDD aplicados al desarrollo web

Durante décadas, el desarrollo de software se centró en escribir código primero y probarlo después. Este enfoque, aunque funcional, generaba un problema recurrente: los errores se descubrían tarde, cuando corregirlos era más costoso. El desarrollo moderno invierte ese orden. Hoy se escribe **primero la prueba y luego el código**, siguiendo el principio de que **la calidad no se inspecciona al final, sino que se diseña desde el inicio**.

A partir de esa idea nacen tres metodologías complementarias:

- **TDD (Test-Driven Development)**
- **ATDD (Acceptance Test-Driven Development)**
- **BDD (Behavior-Driven Development)**

Todas comparten el mismo propósito: garantizar que el software cumple tanto con los requisitos técnicos como con las expectativas del usuario.

TDD: desarrollo guiado por pruebas

El **Test-Driven Development (TDD)** es una metodología en la que las pruebas **se escriben antes del código funcional**. El proceso sigue un ciclo muy claro conocido como **Red – Green – Refactor**:

1. **Red:** escribir una prueba que falle (porque el código aún no existe).
2. **Green:** escribir el código mínimo necesario para que la prueba pase.
3. **Refactor:** mejorar el código manteniendo la prueba en verde.

Este enfoque cambia completamente la mentalidad del desarrollador: ya no se programa "a ciegas", sino que cada línea de código tiene una prueba que la valida.

Ejemplo práctico de TDD con JUnit

Supongamos que estamos desarrollando una función `calcularDescuento()` para una tienda online.

Paso 1 – Red (la prueba falla):

```
@Test  
void testCalcularDescuento() {  
    Tienda tienda = new Tienda();  
    double descuento =  
        tienda.calcularDescuento(100);  
    assertEquals(10, descuento); //  
    Esperamos 10% de descuento  
}
```

Paso 2 – Green (escribir el código mínimo):

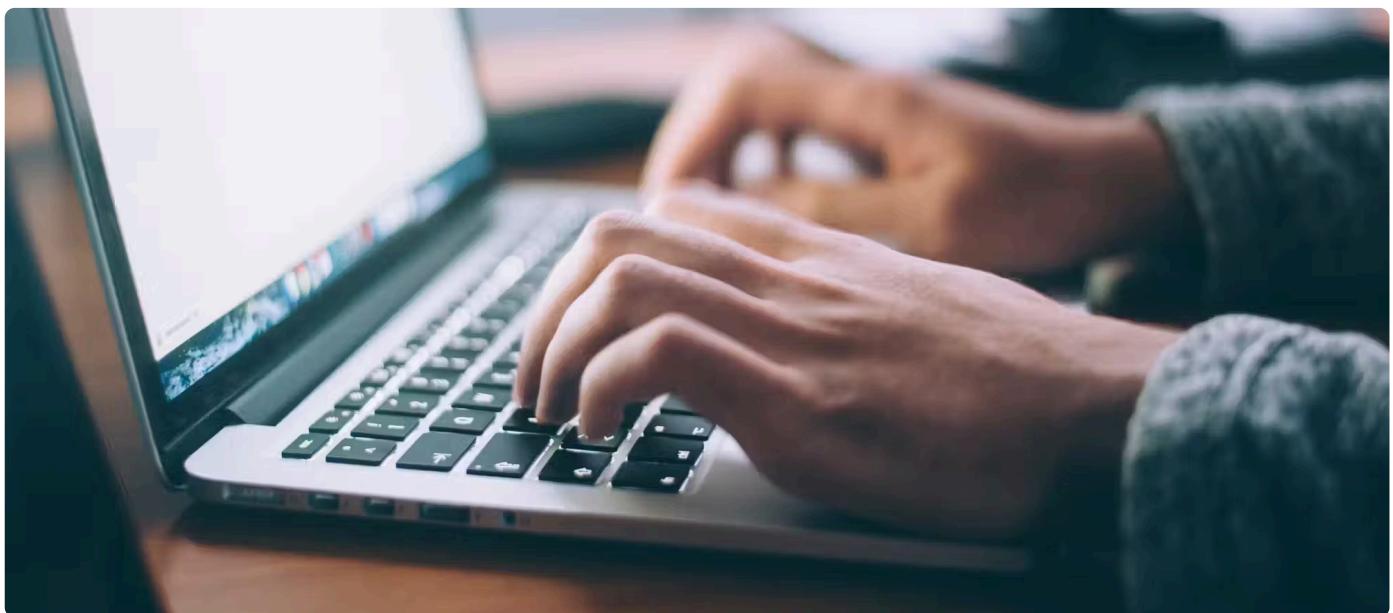
```
public double  
calcularDescuento(double precio) {  
    return precio * 0.10;  
}
```

Paso 3 – Refactor (mejorar sin romper):

Si luego añadimos nuevas reglas (por ejemplo, descuentos por fidelidad), modificaremos el código, pero la prueba original seguirá validando que el comportamiento básico se mantiene.

Resultado

Con cada nueva funcionalidad, añadimos nuevas pruebas, creando un escudo automático contra errores y regresiones.





ATDD: pruebas de aceptación dirigidas por el cliente

El **Acceptance Test-Driven Development (ATDD)** amplía la idea del TDD al nivel del negocio. Aquí, las **pruebas de aceptación** se definen **antes del desarrollo** y en colaboración con el cliente o el responsable funcional del proyecto.

El objetivo es asegurarse de que el software **hace lo que el usuario espera**, no solo que "funciona bien". Los criterios de aceptación se escriben en lenguaje natural o semitécnico, de manera que tanto los desarrolladores como los usuarios los comprendan.

Ejemplo práctico de ATDD

Escenario: un cliente de banca móvil debe poder transferir dinero entre sus cuentas.

Criterio de aceptación:

- Si el usuario tiene saldo suficiente y la cuenta destino es válida, la transferencia debe completarse correctamente.
- Si el saldo es insuficiente, el sistema debe mostrar un mensaje de error.

Antes de escribir una sola línea de código, se documentan estos casos de prueba y se automatizan mediante frameworks como **Cucumber**, **FitNesse** o **Robot Framework**, que permiten vincular estos escenarios con el código real.

De este modo, los desarrolladores programan **siguiendo las expectativas del usuario**, no solo los requerimientos técnicos.

BDD: desarrollo guiado por comportamiento

El Behavior-Driven Development (BDD)

lleva el enfoque de ATDD un paso más allá. Se basa en describir el comportamiento esperado del sistema **en lenguaje natural**, de modo que cualquier persona (técnica o no técnica) pueda entenderlo.

La estructura típica de un escenario BDD es:

- **Given (Dado)**: el contexto inicial.
- **When (Cuando)**: la acción que realiza el usuario.
- **Then (Entonces)**: el resultado esperado.

Ejemplo en desarrollo web

Usando **Cucumber**, podemos definir un escenario para el inicio de sesión en una aplicación web:

Feature: Inicio de sesión de usuario

Scenario: Acceso con credenciales correctas

 Given el usuario "ana@example.com" está registrado

 When introduce su contraseña correcta

 Then accede a su panel de control

Este lenguaje, llamado **Gherkin**, permite traducir requisitos funcionales en pruebas automatizables. Detrás de cada paso ("Given", "When", "Then") hay código Java (u otro lenguaje) que ejecuta acciones reales contra el sistema, validando que se cumple el comportamiento descrito.

Resultado: los requisitos del negocio y las pruebas técnicas se mantienen alineados, eliminando malentendidos entre equipos.

Cómo se integran TDD, ATDD y BDD

Nivel	Enfoque	Participantes	Propósito
TDD	Pruebas unitarias	Desarrolladores	Garantizar que el código funciona correctamente
ATDD	Pruebas de aceptación	Cliente + equipo técnico	Confirmar que se cumple lo que el usuario pidió
BDD	Escenarios de comportamiento	Todo el equipo	Unificar lenguaje entre negocio y desarrollo

Las tres metodologías se aplican de forma escalonada. TDD asegura la **calidad técnica**, ATDD la **calidad funcional**, y BDD la **calidad comunicativa y del comportamiento**.

Esquema visual

El siguiente esquema resume la relación, el enfoque y el uso de las tres metodologías.



ⓘ Descripción del esquema:

- **TDD:** nivel técnico. Define cómo debe comportarse el código desde su base.
- **ATDD:** nivel funcional. Asegura que el sistema cumple las necesidades del usuario.
- **BDD:** nivel comunicativo. Unifica lenguaje y colaboración entre equipos. Los tres enfoques se complementan dentro de un flujo de desarrollo ágil y orientado a calidad.

Caso de estudio

Contexto

BBVA, una de las principales entidades financieras internacionales, buscaba mejorar la coherencia entre sus equipos de negocio y tecnología durante el desarrollo de nuevos servicios en su app móvil. Los requerimientos funcionales se documentaban en lenguaje técnico y los usuarios finales a menudo no los comprendían, generando malentendidos y retrabajos.

Estrategia

El banco implementó un marco de trabajo basado en **BDD** y **ATDD** usando **Cucumber** y **Jenkins**:

01	02	03
Redacción de escenarios Los analistas funcionales redactaban los escenarios de usuario en lenguaje Gherkin .	Implementación automática Los desarrolladores implementaban los pasos en código Java que automatizaban las pruebas.	Ejecución continua Cada commit en el repositorio Git desencadenaba una ejecución automática de las pruebas de comportamiento en el pipeline CI/CD.

Resultado

- Se redujeron los errores funcionales en producción en un **78%**.
- El tiempo de validación de nuevas funcionalidades disminuyó un **45%**.
- Los equipos de negocio y tecnología comenzaron a compartir un **lenguaje común** para describir requisitos.

Conclusión: el uso de BDD permitió que los equipos del BBVA desarrollaran funcionalidades alineadas con la experiencia real del cliente, mejorando la calidad y la velocidad del desarrollo.

Herramientas y consejos



1. Practica el ciclo Red – Green – Refactor

En tu trabajo diario, escribe primero una prueba que falle. Solo después implementa el código necesario para superarla. Este hábito transforma la forma en que razonas sobre los errores y garantiza que cada línea tiene propósito.



2. Usa herramientas adecuadas según el nivel de prueba

- **JUnit** → pruebas unitarias (TDD).
- **Cucumber o Behave** → pruebas BDD.
- **Robot Framework** → pruebas de aceptación (ATDD).

Estas herramientas pueden combinarse dentro del mismo proyecto.



3. Involucra al cliente en ATDD y BDD

No se trata solo de escribir código que funcione, sino de asegurar que **resuelve la necesidad del usuario**. Haz que los escenarios se escriban en lenguaje natural y que el cliente pueda leerlos y validarlos.



4. Integra tus pruebas en el pipeline automático

Conecta tus frameworks de testing a **Jenkins**, **GitHub Actions** o **GitLab CI**. Cada vez que alguien actualiza el repositorio, todas las pruebas se ejecutan automáticamente y los errores se detectan antes de llegar a producción.



5. Documenta los escenarios como conocimiento vivo

Los casos de prueba de BDD se convierten en documentación ejecutable del proyecto. Cada escenario describe cómo debe comportarse el sistema, sirviendo como **manual funcional dinámico**.

Mitos y realidades

 **Mito:** "TDD ralentiza el desarrollo."

→ **FALSO.** Aunque escribir pruebas toma más tiempo al inicio, el ahorro posterior en corrección de errores y retrabajo es mucho mayor. TDD acelera el ciclo completo de entrega.

 **Mito:** "BDD es solo para testers."

→ **FALSO.** El BDD une a desarrolladores, analistas, testers y negocio. No se trata de probar más, sino de **comunicar mejor** y desarrollar con un propósito compartido.

Realidad

Estas metodologías transforman la cultura de desarrollo: el código se valida antes de nacer, y los errores dejan de ser sorpresas para convertirse en parte del proceso de aprendizaje continuo.

Resumen final

- **TDD:** escribir pruebas antes del código (Red → Green → Refactor).
- **ATDD:** definir pruebas de aceptación junto al cliente.
- **BDD:** describir el comportamiento del sistema en lenguaje natural (Given–When–Then).
- **Herramientas:** JUnit, Cucumber, Behave, Robot Framework.
- **Ventaja clave:** menor retrabajo, mejor comunicación y mayor confianza en la calidad del producto.