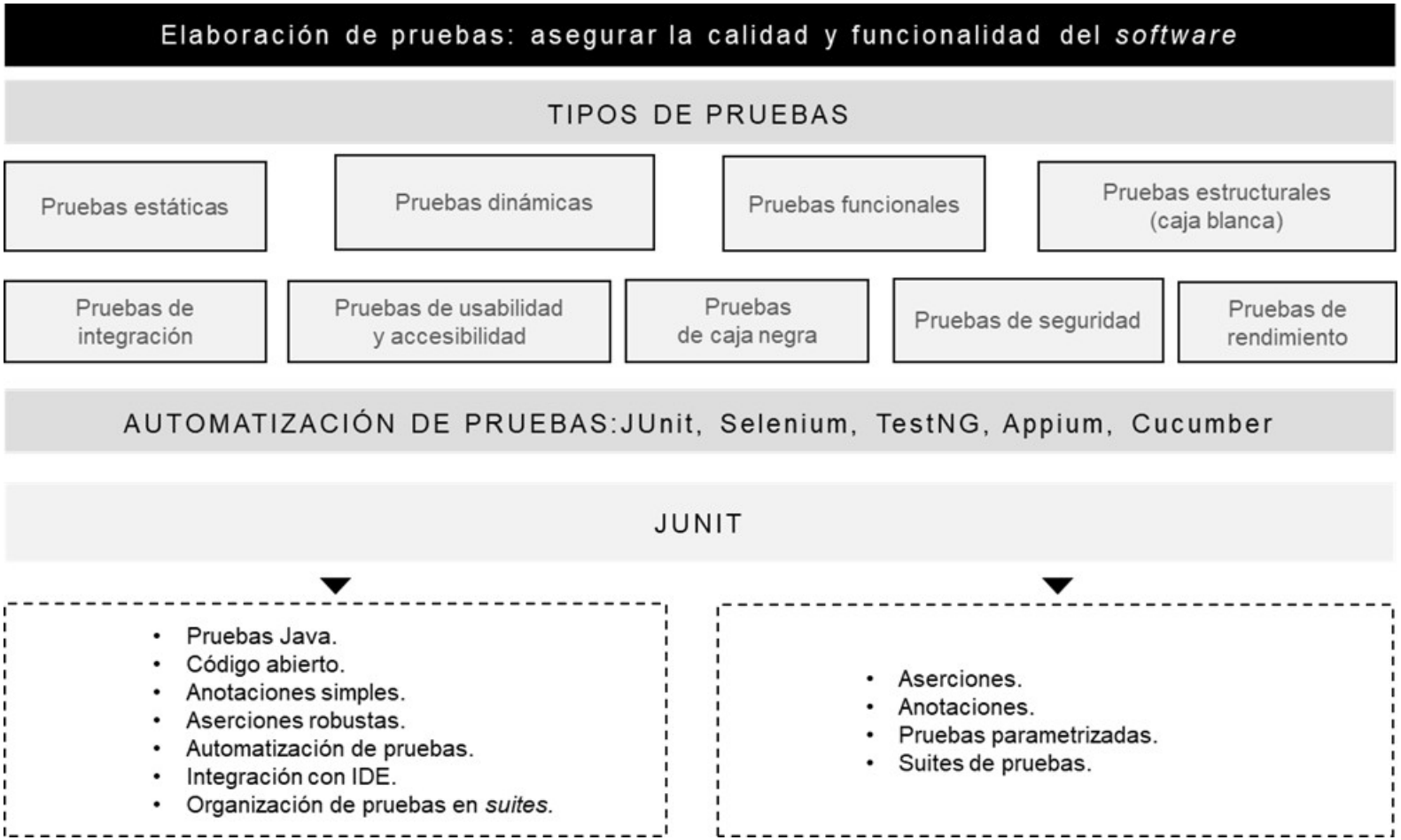
Entornos de Desarrollo

Elaboración de pruebas

Índice

Esquema


Elaboración de pruebas: asegurar la calidad y funcionalidad del *software*
TIPOS DE PRUEBAS
Pruebas estáticas
Pruebas dinámicas
Pruebas funcionales
Pruebas estructurales (caja blanca)
Pruebas de integración
Pruebas de usabilidad y accesibilidad
Pruebas de caja negra
Pruebas de seguridad
Pruebas de rendimiento
AUTOMATIZACIÓN DE PRUEBAS:JUnit, Selenium, TestNG, Appium, Cucumber
JUNIT
• Pruebas Java.
• Código abierto.
• Anotaciones simples.
• Aserciones robustas.
• Automatización de pruebas.
• Integración con IDE.
• Organización de pruebas en *suites*.
• Aserciones.
• Anotaciones.
• Pruebas parametrizadas.
• Suites de pruebas.

Material de estudio

6.1. Introducción y objetivos

En el desarrollo de *software,* la realización de **pruebas** es un componente crítico para asegurar la calidad y la funcionalidad del producto final. Este tema proporciona una visión integral de los diferentes **tipos de pruebas** que se deben llevar a cabo durante el ciclo de vida del desarrollo de una aplicación. Desde pruebas estáticas y dinámicas, hasta pruebas funcionales, estructurales, de integración, usabilidad y accesibilidad, entre otras, cada tipo de prueba tiene un propósito específico y contribuye a la validación y verificación del *software.*

La correcta implementación de estas pruebas no solo garantiza que el *software* cumple con los requisitos técnicos y de usuario, sino que también minimiza los riesgos de fallos en producción, lo que puede tener consecuencias significativas para cualquier organización.

Los **objetivos** principales de este documento son:

- Identificar los **tipos de pruebas** que deben realizarse durante el desarrollo de una aplicación.
- Analizar los **puntos de prueba** de un programa, destacando los métodos y estrategias más efectivos para cada tipo de prueba.
- Diseñar un **plan de pruebas** de una aplicación, proporcionando una guía estructurada para llevar a cabo pruebas exhaustivas.
- Comprender **las características, ámbitos de aplicación** y **estándares** de la automatización de pruebas
- Familiarizarse con las **herramientas** más usadas en la automatización de pruebas.

Estos objetivos buscan equipar a los desarrolladores y testeadores con el conocimiento necesario para implementar un proceso de pruebas robusto y eficiente, que garantice que el *software* no solo cumple con los requisitos establecidos, sino que también ofrece una experiencia de usuario óptima y está preparado para enfrentarse a posibles amenazas y desafíos en su entorno operativo.

6.2. Tipos de pruebas

Entornos de trabajo

En el complejo mundo del desarrollo de *software,* un programa atraviesa **varios entornos** antes de llegar a manos del usuario final. Cada uno de estos entornos cumple un propósito específico y crucial en el ciclo de vida del *software.* En este apartado, exploraremos en detalle los **tres principales** entornos: **desarrollo, preproducción y producción.**

Entorno de desarrollo

El entorno de desarrollo es un lugar donde los programadores pueden trabajar de forma productiva y creativa, concentrándose en el desarrollo del *software* sin preocuparse por las complejidades de la infraestructura.

Dicho entorno está **enfocado al desarrollador,** estableciéndose las configuraciones y medidas de seguridad adecuadas que permitan facilitar y agilizar dicho proceso de desarrollo. Generalmente, el desarrollador tiene permisos de administración sobre el equipo en el que trabaja.

Entorno de preproducción

El entorno de preproducción, también conocido como entorno de pruebas o *staging,* es el **puente** entre el **desarrollo** y la **producción.** Su objetivo principal es **simular** el **entorno de producción** lo más fielmente posible para realizar **pruebas** exhaustivas antes del lanzamiento.

Este es el entorno en donde se realizarán las pruebas oportunas para garantizar que el *software* se comportará de forma adecuada antes de pasar al entorno de producción.

El acceso a la aplicación estará permitido solo para el grupo de testeadores y con datos que, si no son los reales, al menos deberán ser muy semejantes.

Entorno de producción

El entorno de producción es el **destino final del *software,*** donde interactúa con **usuarios reales** y **procesa datos reales.** Es el escenario donde el *software* debe demostrar su valía y cumplir con las expectativas de rendimiento, seguridad y fiabilidad. La versión que se despliegue en este entorno debe haber sido rigurosamente probada previamente en su totalidad.

Siempre debe existir un **plan de reversión** *(rollback)* en caso de problemas críticos. Este plan debe ser probado y el equipo debe estar familiarizado con su ejecución.

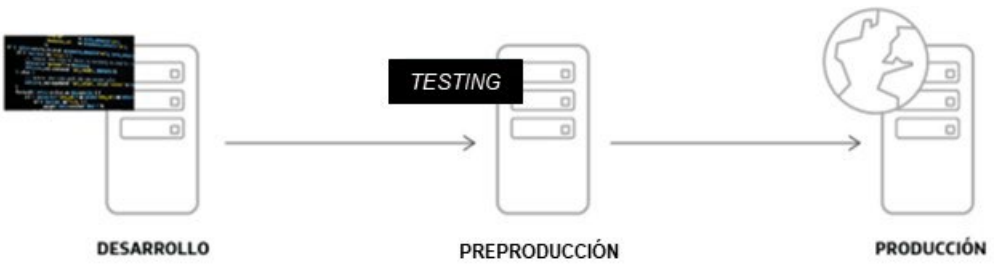


Figura 1. Entornos de trabajo. Fuente: elaboración propia.

El proceso de pruebas

El proceso de pruebas es una fase crucial en el ciclo de vida del desarrollo de *software.* Su objetivo principal es demostrar **que la aplicación funciona según lo diseñado,** cumpliendo con los requisitos establecidos y las expectativas del cliente. Este proceso no solo busca verificar la funcionalidad, sino también validar la calidad, el rendimiento y la seguridad del *software.*

- **Cumplimiento de requisitos:** las pruebas deben verificar que el *software* cumple con los requisitos del cliente (necesidades del usuario) requisitos técnicos (el *software* cumple con las especificaciones técnicas) y los requisitos de negocio (cumple los objetivos y procesos de negocio).

- **Funcionamiento adecuado:** el proceso de pruebas busca identificar fallos, validar el comportamiento esperado y evaluar que el *software* opera de manera eficiente bajo diferentes condiciones de carga.

Validación y verificación

Aunque a menudo se confunden, la validación y la verificación son procesos distintos en las pruebas de *software:*

Validación

- **Pregunta clave:** ¿estamos construyendo el producto correcto?
- **Enfoque:** asegura que el *software* satisface las necesidades reales del cliente.
- **Objetivo:** confirmar que el sistema cumple con el propósito previsto en el mundo real.

Verificación

- **Pregunta clave:** ¿estamos construyendo el producto correctamente?
- **Enfoque:** evalúa si el *software* está bien diseñado y libre de errores.

- **Objetivo:** garantizar que el sistema cumple con las especificaciones técnicas y de diseño.

Es crucial entender que un *software* puede pasar todas las verificaciones técnicas (funcionar sin errores) y aun así fallar en la validación si no satisface las necesidades reales del usuario final. Por lo tanto, ambos procesos son esenciales para el éxito del proyecto.

Tipos de pruebas

En el ámbito del desarrollo de *software,* las pruebas se dividen en dos categorías principales: **pruebas estáticas** y **pruebas dinámicas.**

Las **pruebas estáticas** son aquellas que se realizan **sin ejecutar el código** de la aplicación. Se centran en la revisión y análisis de los documentos de diseño, código fuente y otros elementos del *software:*

- Se realizan en las etapas tempranas del desarrollo lo que reduce los costes de corrección.
- No requieren la ejecución del código.
- Son efectivas para detectar defectos en la fase de diseño y codificación.

Este tipo de pruebas mejora la calidad del código y la documentación y fomenta el intercambio de conocimientos entre el equipo de desarrollo.

Por otro lado, las **pruebas dinámicas** implican la **ejecución real del código** de la aplicación. Se centran en el comportamiento del *software* durante su funcionamiento:

- Requieren la ejecución del código o de una parte del sistema.
- Se realizan en diferentes niveles: unidad, integración, sistema y aceptación.
- Evalúan el comportamiento real del *software* en diversas condiciones.

Las pruebas dinámicas verifican el comportamiento real del *software* en ejecución y permiten evaluar aspectos como rendimiento, usabilidad y seguridad.

Pruebas funcionales

Las pruebas funcionales son una técnica de **evaluación de *software*** que se centra en verificar si el sistema cumple con los requisitos y especificaciones funcionales sin necesidad de conocer su lógica interna y que este responde correctamente a las entradas proporcionadas.

El objetivo principal de las pruebas funcionales es **confirmar que el *software* se comporta como se describe en sus especificaciones,** asegurando que cumple con los requisitos funcionales definidos.

Pruebas estructurales

Las pruebas estructurales, también conocidas como pruebas de caja blanca o caja de cristal, implican evaluar un sistema con un conocimiento completo de su estructura, diseño e implementación. A diferencia de las pruebas funcionales, que se centran en el comportamiento del sistema desde una perspectiva externa, las pruebas estructurales analizan **cómo funciona el *software* internamente.**

En este tipo de pruebas, se elige una entrada específica y se examina la aplicación a través de los diferentes caminos posibles en el código. El objetivo es verificar las salidas generadas y asegurar que el *software* se comporta como se espera en función de su estructura interna.

Las pruebas estructurales son **aplicables a diversas fases de evaluación:**

- **Pruebas unitarias:** se prueban rutas específicas dentro de una unidad de código.
- **Pruebas de integración:** se examinan los caminos entre diferentes unidades del sistema.
- **Pruebas de sistema:** se revisan los caminos entre los subsistemas del *software.*

Ventajas:

- **Inicio temprano:** se pueden realizar tan pronto como el código está disponible, sin necesidad de tener la interfaz gráfica completa.
- **Cobertura completa:** permite verificar exhaustivamente todos los posibles caminos a través del código.

Desventajas:

- **Requiere conocimientos técnicos profundos:** es necesario tener un entendimiento detallado del código fuente y de la programación, lo cual puede ser complejo.
- **Modificaciones constantes:** los *scripts* de prueba pueden necesitar ajustes frecuentes si el código cambia regularmente.

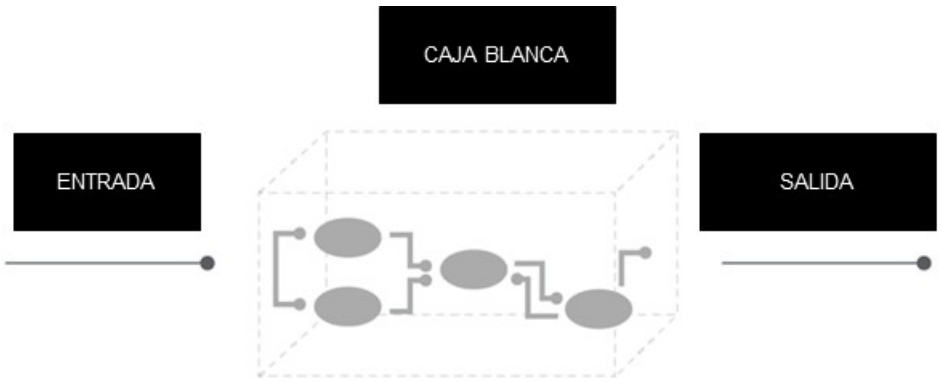


Figura 2. Pruebas estructurales. Fuente: elaboración propia.

Pruebas de integración

Las pruebas de integración se realizan tras las pruebas unitarias y antes de las pruebas de sistema y se centran en **evaluar cómo interactúan entre sí las unidades** independientes de un sistema una vez combinadas. Estas pruebas se llevan a cabo para identificar errores que puedan surgir debido a la interacción entre los distintos componentes del sistema.

El **objetivo** principal es verificar que:

- Cada componente interactúa correctamente con sus interfaces.
- Cada unidad cumple con su funcionalidad individualmente.
- El sistema en conjunto opera de manera adecuada y cumple con los requisitos funcionales globales.

Pruebas de usabilidad y accesibilidad

Las pruebas de usabilidad y accesibilidad se centran en la **facilidad de uso** y la **inclusión de una aplicación.** Estas pruebas son cruciales para garantizar que el *software* no solo sea fácil de utilizar, sino también accesible para todos los usuarios, independientemente de sus capacidades.

La **usabilidad** se refiere a qué tan clara, fácil, efectiva y satisfactoriamente los usuarios pueden interactuar con una aplicación. Durante estas pruebas, se les plantean una serie de preguntas para obtener sus opiniones y evaluar su experiencia.

La **accesibilidad,** según la W3C, implica que un sitio web debe ser **accesible de manera universal** asegurando que la aplicación se adapte a diversas discapacidades, como problemas visuales, auditivos, de motricidad o cognitivos, permitiendo que todos los usuarios puedan acceder a la información y funcionalidades ofrecidas.

Pruebas de caja negra

Las pruebas de caja negra son una técnica de evaluación del *software* en la que se **examinan** las **funcionalidades** del sistema **sin conocer su estructura interna.** En otras palabras, el tester interactúa con el *software* como si fuera una caja negra, sin acceso al código o a los detalles de implementación.

Estas pruebas, aunque, generalmente, se enfocan en **aspectos funcionales,** también pueden abordar **aspectos no funcionales,** como el rendimiento del sistema.

Objetivos de las pruebas de caja negra:

- Identificar funciones incorrectas o ausentes.
- Detectar errores en la interfaz de usuario.
- Localizar errores en el acceso a los datos.
- Detectar fallos en el comportamiento general del *software.*
- Verificar errores en el inicio y la terminación de la aplicación.

Ventajas de las pruebas de caja negra:

- **Detección de no conformidades:** permite identificar discrepancias con las especificaciones desde la perspectiva del usuario.
- Requisitos de Perfiles: No es necesario contar con conocimientos especializados en programación o implementación del software para realizar las pruebas.
- Perspectiva Objetiva: Al ser realizadas por personal ajeno a los programadores, las pruebas ofrecen una visión imparcial del sistema.
- Desarrollo Temprano de Casos de Prueba: Los casos de prueba pueden elaborarse tan pronto como se definen las especificaciones.

Desventajas de las pruebas de caja negra:

- **Cobertura incompleta:** algunas rutas de ejecución pueden no ser probadas, ya que las pruebas se basan en un conjunto limitado de entradas.
- **Dificultad en el diseño de pruebas:** puede resultar complicado diseñar pruebas adecuadas si las especificaciones no están bien definidas.

Pruebas de seguridad

Las pruebas de seguridad tienen como propósito garantizar que el ***software*** esté **protegido contra amenazas** tanto externas como internas, que pueden provenir de personas o programas maliciosos. Estas pruebas se llevan a cabo por técnicos especializados con formación específica en seguridad de *software.* La inclusión de pruebas de seguridad en el proceso de desarrollo es altamente recomendada para minimizar riesgos.

En términos generales, estas pruebas se enfocan en **verificar** varios **aspectos cruciales** del *software.* Entre ellos, se evalúa si el sistema cuenta con una autenticación adecuada, si el control de acceso está correctamente implementado, si es capaz de mantener la confidencialidad de los datos y si puede garantizar su disponibilidad frente a posibles ataques por parte de *hackers* o *software* malicioso.

Pruebas de rendimiento

En el desarrollo de *software,* a menudo es difícil prever los valores máximos que una aplicación podrá soportar o cómo se comportará bajo situaciones de carga intensa. Por esta razón, se realizan pruebas de comportamiento para **evaluar** el **rendimiento** de la aplicación.

Estas pruebas se llevan a cabo en entornos de preproducción que replican lo más fielmente posible el entorno de producción. Utilizando herramientas especializadas, se simulan las condiciones de carga necesarias, como el número de usuarios concurrentes.

Los principales **parámetros** que se obtienen de estas pruebas incluyen:

- **Tiempos de respuesta:** este es uno de los indicadores más importantes en las pruebas de rendimiento, ya que afecta directamente la experiencia del usuario con la aplicación.

- **Uso de recursos:** se reporta de manera descriptiva, por ejemplo, «durante las pruebas, la utilización de la CPU no superó el 60 % en ningún momento».

- **Volúmenes, capacidades y ratios:** incluyen métricas como el ancho de banda utilizado, transacciones por segundo, número de usuarios, entre otros.

Existen **dos tipos principales de pruebas** para verificar el rendimiento del sistema:

- **Pruebas de carga:** su objetivo es identificar la capacidad máxima de la aplicación y detectar posibles cuellos de botella que puedan afectar su rendimiento. Se somete a la aplicación a una carga de solicitudes que se estima como la máxima que podría soportar en condiciones reales.

- **Pruebas de estrés:** estas pruebas se centran en evaluar la robustez, disponibilidad y estabilidad de la aplicación bajo condiciones extremas. Se busca observar el comportamiento del sistema en situaciones que pueden no coincidir con las condiciones normales de operación, como denegaciones de servicio, tiempos de espera prolongados, mensajes de error o corrupción de datos.

Pruebas de *software*

Las pruebas de *software* son un componente crucial en el proceso de desarrollo, destinado a asegurar que las aplicaciones funcionen de acuerdo con las especificaciones y satisfagan las expectativas del usuario final. Estas pruebas se dividen en varias categorías, cada una con un enfoque y objetivos específicos para abordar diferentes aspectos del *software:*

- **Pruebas de unidad:** estas pruebas se centran en verificar el correcto funcionamiento de las unidades individuales de código, como funciones, métodos o clases. Su objetivo principal es identificar errores en las unidades de código antes de integrarlas en el sistema más amplio. **Tipos:**
    - **Caja negra:** conociendo la función específica para la que fue diseñado el producto. Enfoque funcional.
    - **Caja blanca:** conociendo el funcionamiento del producto. Enfoque estructural.
    - **Conjetura de errores:** se listan los errores comunes y se diseñan casos de prueba en función de estos.
    - **Enfoque aleatorio:** se simula la entrada de datos en el sistema con generadores de pruebas.

- **Pruebas de integración:** una vez que las unidades de código han sido probadas de manera individual, las pruebas de integración se encargan de evaluar cómo interactúan entre sí. Estas pruebas son cruciales para detectar problemas en la comunicación y cooperación entre diferentes módulos del *software,* asegurando que los componentes funcionen correctamente cuando se combinan. Tipos:
    - **Pruebas de regresión:** buscan reducir los efectos colaterales de añadir un nuevo componente al sistema. Hay **tres clases:** pruebas sobre todas las funciones, sobre las afectadas por el cambio o sobre los componentes que han cambiado.
    - **Pruebas de humo:** buscan cubrir la mayor parte de la funcionalidad del sistema. Es una estrategia de integración continua.

- **Pruebas de sistema:** estas pruebas examinan el sistema completo como un todo, asegurando que todas las partes del *software* funcionen juntas de manera armoniosa. Se realizan para validar que el sistema cumple con los requisitos funcionales y no funcionales especificados. Las pruebas de sistema son esenciales para verificar la integración de todos los componentes y su comportamiento en un entorno que simula el entorno de producción. **Tipos:**
    - **Prueba de recuperación:** se fuerza el fallo del sistema para verificar que sabe recuperarse del error en un tiempo determinado (MTTR).
    - **Prueba de seguridad:** busca comprobar que el sistema sabe protegerse de accesos indeseados o ilegales.
    - **Prueba de resistencia *(test stress):*** se aplica para ver cómo se comporta el sistema ante situaciones extremas.
    - **Prueba de rendimiento:** sirve para probar el rendimiento del *software* en tiempo de ejecución en un sistema integrado.

- **Pruebas de seguridad:** la creación de *software* seguro requiere una comprensión básica de los principios de seguridad. El objetivo de la seguridad del *software* es mantener la confidencialidad, integridad y disponibilidad de la información recursos para permitir operaciones comerciales exitosas. Este objetivo se logra a través de la implementación de controles de seguridad. Los fallos de seguridad del

software se pueden introducir en cualquier etapa del ciclo de vida del desarrollo del *software,* lo que incluye:

- No identificar los requisitos de seguridad por adelantado.
- Crear diseños conceptuales que tienen errores lógicos.
- Usar malas prácticas de codificación que introducen vulnerabilidades técnicas.
- Implementar el *software* incorrectamente.
- Introducir fallas durante el mantenimiento o actualización.
- Además, es importante entender que las vulnerabilidades del *software* pueden tener un alcance más allá del *software* en sí mismo.

Es importante que los equipos de desarrollo web comprendan que los controles del lado del cliente, como la entrada basada en la validación del lado del cliente, los campos ocultos y controles de interfaz (por ejemplo, menús desplegables y botones de radio) brindan poco o ningún beneficio de seguridad.

- **Pruebas de validación:** las pruebas de validación se centran en comprobar que el *software* cumpla con las necesidades y expectativas del usuario final. Estas pruebas se realizan para verificar que el *software* no solo funcione correctamente, sino que también cumpla con los requisitos establecidos y entregue la funcionalidad esperada en el contexto real de uso. **Tipos:**
  - **Revisión de la configuración:** también llamada auditoría. Comprueba que todos los elementos de la configuración SW se han desarrollado bien.
  - **Pruebas alfa y beta:** son pruebas de aceptación, para ver cómo el usuario utiliza realmente el programa. La **prueba alfa** la realiza un cliente en un entorno controlado y el desarrollador está presente. La **prueba beta** la realizan los usuariosfinales en sus lugares de trabajo. El desarrollador no está y se le envían informes.

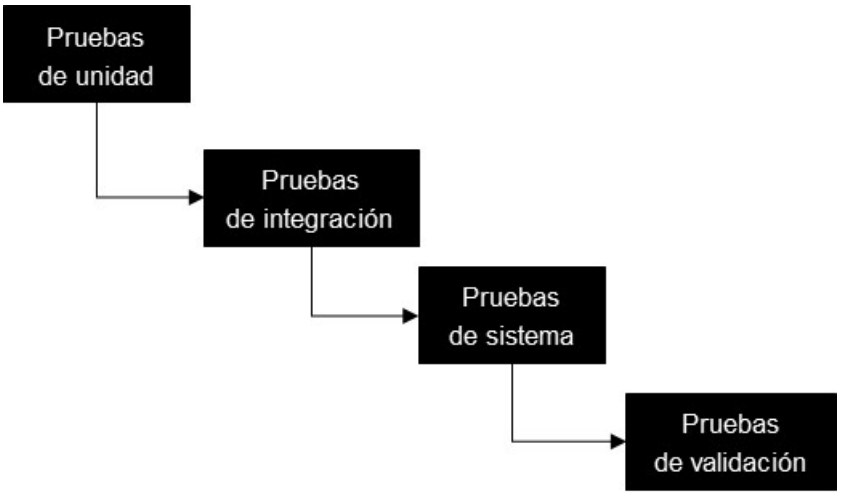


Figura 3. Pruebas de *software.* Fuente: elaboración propia.

Cada una de estas pruebas juega un papel fundamental en el ciclo de desarrollo de *software,* contribuyendo a la creación de aplicaciones robustas y fiables que satisfagan las expectativas de los usuarios y cumplan con los estándares de calidad.

Planificación de las pruebas

La **planificación de las pruebas** es una etapa crucial en el proceso de aseguramiento de calidad del *software.* Implica una **programación detallada** y una **estimación** de los **recursos** necesarios para llevar a cabo las pruebas de manera efectiva. Esta fase prevé aspectos clave como la **definición de estándares,** la **descripción** de las **pruebas** y la **asignación de recursos.**

Aspectos fundamentales en la planificación de pruebas

- **Proceso de prueba:** se debe detallar cómo se llevarán a cabo las principales fases del proceso de pruebas del sistema. Esto puede incluir pruebas de subsistemas individuales, pruebas de interfaces externas y otras áreas específicas según el alcance de las pruebas.

- **Requisitos de trazabilidad:** la planificación debe garantizar que todos los requisitos del *software* sean probados de manera individual, asegurando que cada requisito esté cubierto por las pruebas correspondientes.
- **Elementos para probar:** especificar todos los elementos del software que serán objeto de pruebas, detallando los componentes y funciones que se evaluarán.

- **Cronograma de pruebas:** establecer un calendario de pruebas y una asignación de recursos que esté alineado con el cronograma general del proyecto, asegurando que las pruebas se realicen en el momento adecuado.

Componentes claves de un plan de pruebas efectivo

Un plan de pruebas efectivo es fundamental para garantizar la calidad y el rendimiento de cualquier *software* o sistema. Este documento estratégico debe incluir elementos esenciales como:

- **Procedimientos de registro de prueba:** el proceso de pruebas no se limita a la ejecución; los resultados deben ser registrados de forma sistemática. Además, el proceso debe ser auditable para verificar su correcta realización.

- **Requisitos de *hardware* y *software:*** esta sección debe detallar los recursos de *software* y *hardware* necesarios para llevar a cabo las pruebas, permitiendo su planificación anticipada por parte de los administradores de sistemas.

- **Plan de contingencias:** debe incluir estrategias para manejar imprevistos que puedan afectar el proceso de pruebas, como la falta de personal o problemas con el *software.*

- **Definición de los casos de prueba:** especificar los casos de prueba que se aplicarán al sistema, derivados de la especificación de requisitos. Estos casos de prueba deben cubrir todos los aspectos críticos del sistema para asegurar una evaluación exhaustiva.

Tipo de prueba	Características	Ventajas
Pruebas de unidad	Verifica componentes individuales del código.	Detecta errores tempranos, mejora la calidad del código.
Pruebas de integración	Evalúa la interacción entre módulos.	Identifica problemas de integración, asegura cohesión.
Pruebas de sistemas	Prueba el sistema completo en un entorno simulado.	Verifica la funcionalidad general y la integración total.
Pruebas de validación	Asegura que el *software* cumple con las expectativas del usuario.	Garantiza que el *software* satisface las necesidades del usuario.
Pruebas de seguridad	Identifica vulnerabilidades y riesgos de seguridad.	Protege contra ataques, asegura integridad de datos.
Pruebas de rendimiento	Evalúa el comportamiento bajo cargas específicas.	Identifica cuellos de botella, asegura escalabilidad.
Pruebas de usabilidad	Evalúa la facilidad de uso y la experiencia del usuario.	Mejora la experiencia del usuario, identifica el problema de interfaz.
Pruebas funcionales	Verifica que las funciones operen según especificaciones.	Asegura que todas las funciones cumplen con los requisitos.
Pruebas estructurales	Evalúa la estructura interna del *software*.	Identifica errores en la lógica de implementación, mejora la cobertura de pruebas.

Tabla 1. Tipos de pruebas. Fuente: elaboración propia.

6.3. Herramientas de *testing*

Automatización de pruebas

La automatización de pruebas es una práctica esencial en el desarrollo de *software* moderno, que implica el uso de herramientas especializadas para **ejecutar pruebas automáticamente, comparar** los **resultados obtenidos** con los esperados y generar informes detallados. Este enfoque permite una ejecución de pruebas más eficiente, repetitiva y menos propensa a errores humanos, lo cual es crítico en proyectos de gran envergadura o en metodologías ágiles.

La automatización de pruebas ofrece varias **ventajas clave.** En primer lugar, mejora la **eficiencia** y **velocidad,** ya que permite ejecutar numerosas pruebas en poco tiempo. Además, asegura **cobertura** y **consistencia,** abarcando una amplia gama de casos de prueba y garantizando que se realicen de manera uniforme y sin omisiones. Aunque la implementación inicial puede ser costosa, la reducción de costes a largo plazo se logra gracias al **ahorro** de **tiempo** y **esfuerzo** humano. Finalmente, la automatización facilita la **detección temprana de errores,** permitiendo identificar y corregir defectos en etapas tempranas del desarrollo, lo que reduce, significativamente, los costes asociados a errores tardíos.

Herramientas para la automatización de pruebas

El mercado ofrece una gran variedad de herramientas para automatizar pruebas, muchas de ellas especializadas según el lenguaje de programación o el tipo de pruebas a realizar. A continuación, se presentan algunas de las más destacadas:

- **JUnit:** una herramienta ampliamente utilizada en la automatización de pruebas unitarias en aplicaciones Java.
- **Selenium:** un *framework* popular para la automatización de pruebas de interfaces de usuario en aplicaciones web.
- **TestNG:** similar a JUnit, pero con funcionalidades adicionales que facilitan la creación de suites de pruebas más complejas.
- **Appium:** ideal para la automatización de pruebas en aplicaciones móviles (iOS y Android).

- **Cucumber:** un marco de trabajo que permite realizar pruebas basadas en comportamiento (BDD), donde los casos de prueba se escriben en un lenguaje natural como Gherkin.

En la sección A fondo se encuentra el artículo «¿Cómo seleccionar la mejor herramienta de QA?» sobre las distintas herramientas de *testing* y sus características.

Pruebas unitarias y JUnit

Las **pruebas unitarias** son un tipo de prueba que se enfoca en **verificar** el **correcto funcionamiento** de las unidades más pequeñas de un *software,* como funciones o métodos individuales. Estas pruebas son esenciales para asegurar que cada componente del código funcione según lo esperado de manera aislada, antes de integrarlo con otros módulos del sistema.

JUnit es un marco de trabajo ampliamente utilizado en la automatización de pruebas unitarias en Java. Proporciona un conjunto de herramientas y bibliotecas que permiten a los desarrolladores escribir, ejecutar y gestionar pruebas de manera eficiente. JUnit facilita la comparación de los resultados obtenidos con los esperados, lo que ayuda a identificar rápidamente cualquier discrepancia o error en el código.

Entre las principales características de JUnit se encuentran las **aserciones,** que permiten verificar las condiciones esperadas, y las **anotaciones,** que simplifican la configuración y ejecución de las pruebas. JUnit también soporta la ejecución automatizada de pruebas en entornos de desarrollo integrados (IDE) como Eclipse o NetBeans, y su integración en procesos de integración continua, lo que permite una validación continua del código durante todo el ciclo de desarrollo.

Utilización de JUnit

JUnit es un *framework* esencial para realizar pruebas unitarias en Java, ampliamente utilizado por desarrolladores para asegurar la calidad y fiabilidad del código. Entre sus características más destacadas se encuentran:

- **Código abierto:** JUnit es un *framework* de código abierto, lo que facilita su acceso y modificación según las necesidades del proyecto.

- **Anotaciones simples:** JUnit proporciona una serie de anotaciones (@Test, @Before, @After, entre otras) que permiten definir y organizar pruebas.

- **Aserciones robustas:** ofrece una variedad de métodos de aserción (assertEquals, assertTrue, assertFalse, etc.) que permiten verificar si el resultado de un test es el esperado, facilitando la detección de fallos en el código.

- **Automatización de pruebas:** JUnit permite la ejecución automática de pruebas, proporcionando retroalimentación inmediata y reduciendo significativamente el tiempo dedicado a la validación manual.

- **Integración con IDE y herramientas de construcción:** JUnit se integra fácilmente con entornos de desarrollo integrados (IDE) como Eclipse y NetBeans, así como con herramientas de construcción como Maven y Gradle, permitiendo la ejecución continua de pruebas.

- **Organización de pruebas:** soporta la agrupación de pruebas en *suites,* lo que facilita la ejecución conjunta de múltiples casos de prueba relacionados.

¿Cómo Funciona JUnit?

El funcionamiento de JUnit se basa en la ejecución de métodos de prueba definidos dentro de una clase. Cada método de prueba está identificado por la anotación **`@Test`.** Al ejecutarse, JUnit comprueba si las condiciones especificadas en las

aserciones son verdaderas. Si todas las aserciones pasan, la prueba se considera exitosa; de lo contrario, se marca como fallida.

En la sección A fondo se muestra en el vídeo *Cómo configurar y crear tests de JUnit 5 en Eclipse* y *Cómo configurar y crear tests de JUnit 5 en NetBeans.*

La clase JUnit

Para definir un caso de prueba en Java, creamos una clase con métodos. Cada método dentro de esta clase que tenga la anotación **@Test,** representa una prueba específica a ejecutar.

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class CuentaTest  {

    @Test
    public void testCuentaNueva () {
        Cuenta cuenta = new Cuenta();
        Assertions.assertEquals(cuenta.getSaldo(), 0.00);
    }

     @Test
    public void testIngreso () {
        Cuenta cuenta = new Cuenta();
        cuenta.ingresar(100.00);
        Assertions.assertEquals(cuenta.getSaldo(), 100.00);
    }

}
```

Figura 4. Ejemplo de clase JUnit con métodos. Fuente: elaboración propia.

En este ejemplo se utiliza la función **assertEquals()** para verificar que el saldo inicial de la cuenta es una cantidad determinada (cero en el primer método, cien en el segundo). Si el saldo es el esperado, la prueba pasa; de lo contrario, falla.

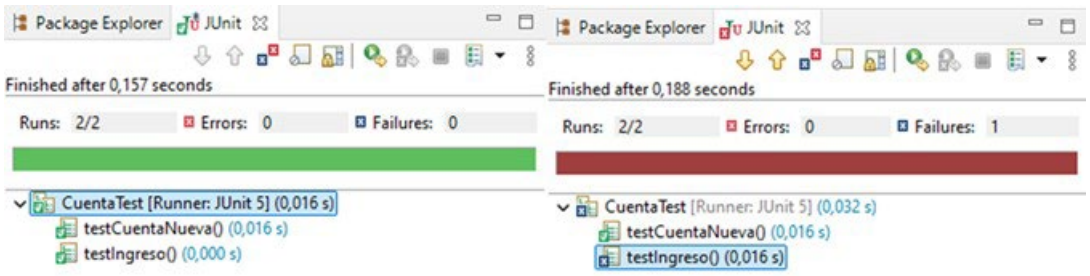


Figura 5. Ejemplo de prueba correcta y fallida. Fuente: elaboración propia.

Métodos de Junit

Los métodos *assert* utilizados por JUnit son:

- **`assertArrayEquals()`:** verifica que dos *arrays* sean iguales en contenido y orden.
- **`assertEquals()`:** comprueba que dos valores sean iguales.
- **`assertTrue()`:** asegura que una condición sea verdadera.
- **`assertFalse()`:** asegura que una condición sea falsa.
- **`assertNull()`:** verifica que un objeto sea *null.*
- **`assertNotNull()`:** verifica que un objeto no sea *null.*
- **`assertSame()`:** verifica que dos referencias apunten al mismo objeto.
- **`assertNotSame()`:** verifica que dos referencias no apunten al mismo objeto.
- **`assertAll()`:** permite agrupar varias aserciones en una sola llamada

A continuación, vamos a ver algunos **ejemplos** de uso de los métodos de JUnit:

```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class PersonaTest {

    @Test
    public void testCompararStringArray() {

        String[] arrayEsperado = {"Juan", "Mar□a", "Jose"};
        String[] arrayPrueba    = {"Juan", "Mar□a", "Jose"};

        // compara el array de prueba con el array principal, determinando si son iguales
        // lo que significa que el resultado es el esperado
        Assertions.assertArrayEquals(arrayEsperado, arrayPrueba);
    }

    @Test
    public void testComprarIgual() {
        // compara los dos datos y determina si el dato resultante tiene el valor deseado
        Assertions.assertEquals("datoresultante", "dato");
    }


    @Test
    public void testComprarTrue() {

        // con assertTrue() y assertFalse() validamos si un resultado es verdadero o falso.
        Assertions.assertTrue(true);
    }


}
```

Figura 6. Utilización de métodos *assert.* Fuente: elaboración propia.

Anotaciones JUnit

Para definir un método de prueba dentro de una clase de pruebas en JUnit, es necesario anotarlo con **@Test.** Esta anotación indica al *framework* que el método en cuestión debe ser ejecutado como una prueba.

JUnit ofrece diversas **anotaciones adicionales** para personalizar la ejecución de las pruebas:

- **@RunWith:** permite especificar un ejecutor de pruebas personalizado en lugar del predeterminado.
- **@Before:** indica que el método debe ejecutarse antes de cada método de prueba. Existen varias anotaciones @Before y alguna de ellas requiere que el método sea estático.

- **@After:** especifica que el método debe ejecutarse después de cada método de prueba. Existen varias anotaciones @After y alguna de ellas requiere que el método sea estático.

```
@BeforeAll
public static void iniciar () {
    System.out.println("Iniciando pruebas ");
}

@AfterAll
public static void terminar () {
    System.out.println("pruebas finalizadas ");
}
```

Figura 7. Utilización de notaciones. Fuente: elaboración propia.

En la sección A fondo se muestras en el vídeo *Cómo usar BeforeEach, AfterEach, BeforeAll y AfterAll en JUnit 5* ejemplos de utilización de las anotaciones de JUnit

---

Suites de pruebas

A medida que crece la complejidad de un programa, aumenta la necesidad de **agrupar casos de prueba** para optimizar su ejecución. En este ejemplo, hemos creado una *suite* con todas las clases que queremos probar y que hemos creado en apartados anteriores.

```
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({ CuentaTest.class, PersonaTest.class })
public class AllTests {

}
```

Figura 8. JUnit *suite.* Fuente: elaboración propia.

Pruebas parametrizadas

Un solo dato puede no ser suficiente para verificar la robustez de un método. Es fundamental probarlo con **diversos escenarios y datos de entrada.** Con JUnit

podemos **parametrizar pruebas,** enviando mediante una lista una serie de datos a comprobar.

```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

public class CuentaTestParametrizado {

    private Cuenta cuenta;
    private double saldoInicial;
    private double deposito;
    private double saldoEsperado;

    // indica que es un test parametrizado y los valores de
    // la prueba ( saldo inicial, ingreso, saldo esperado)
    @ParameterizedTest
    @CsvSource({
        "100.0, 50.0, 150.0",
        "200.0, 75.0, 275.0",
        "300.0, 100.0, 400.0"
    })
    public void testDeposito(double saldoInicial, double deposito, double saldoEsperado ) {

        cuenta = new Cuenta(saldoInicial);
        cuenta.ingresar(deposito);
        Assertions.assertEquals(saldoEsperado, cuenta.getSaldo(), 0.001,
                "El saldo después de ingresar " + deposito + " debería ser " + saldoEsperado);
    }
}
```

Figura 9. JUnit test parametrizado. Fuente: elaboración propia

En la sección A fondo tienes un enlace al vídeo *Curso de JUnit 5 (completo)* de un curso completo de JUnit5.

A fondo

Herramientas de Testing y sus características

EmployIT. (2023, septiembre 8). *¿Cómo seleccionar la mejor herramienta de QA?* Linkedin. https://www.linkedin.com/pulse/c%C3%B3mo-seleccionar-la-mejor-herramienta-de-qa-employit/

En este artículo se presenta una variedad de herramientas de *testing,* adaptadas a los diferentes tipos de pruebas de *software,* indicando su funcionalidad principal.

Curso completo de JUnit5

Makigas. (2023). *Curso de JUnit 5 (completo)* [Vídeo]. YouTube. https://www.youtube.com/playlist?list=PLTd5ehIj0goPcVH3xhSudzyazW8CtMvsq

En este curso se muestran las técnicas avanzadas y mejores prácticas para escribir pruebas de alta calidad utilizando JUnit5. Enseña cómo usar las anotaciones, los métodos de test y las parametrizaciones de pruebas.

Cómo configurar y crear test de JUnit 5 en Eclipse

Makigas. (2023). *Cómo configurar y crear tests de JUnit 5 en Eclipse* [Vídeo]. YouTube. https://www.youtube.com/watch?v=wkXL3emg-NU

Este vídeo muestra cómo configurar y crear test de JUnit5 en Eclipse.

Cómo configurar y crear test de JUnit 5 en Netbeans

Makigas. (2023, marzo 10). *Cómo configurar y crear tests de JUnit 5 en Netbeans* [Vídeo]. YouTube. https://www.youtube.com/watch?v=4wdQ_Xdg_l0

Este vídeo muestra cómo configurar y crear test de JUnit5 en NetBeans.

Cómo usar BeforeEach, AfterEach, BeforeAll y AfterAll en JUnit 5

Makigas. (2023, marzo 8). *Cómo usar BeforeEach, AfterEach, BeforeAll y AfterAll en JUnit 5* [Vídeo]. YouTube. https://www.youtube.com/watch?v=CQDZilNB1PA&t=305s

Este vídeo muestra ejemplos de utilización de las anotaciones de Junit.

Entrenamientos

Entrenamiento 1

► Planteamiento del ejercicio

Compara las pruebas de caja negra y las pruebas de caja blanca.

► Desarrollo paso a paso

1. Define qué son las pruebas de caja negra y las pruebas de caja blanca.

2. Identifica las principales diferencias entre ambas.

3. Explica las ventajas y desventajas de cada tipo de prueba.

► Solución

Pruebas de caja negra:

► **Definición:** evaluación del *software* sin conocer su estructura interna.

► **Ventajas:**

- No requiere conocimientos técnicos profundos.
- Permite detectar errores en la interfaz de usuario y en la funcionalidad general.
- Casos de prueba pueden ser diseñados desde las primeras etapas del desarrollo.

► **Desventajas:**

- Cobertura incompleta de rutas de ejecución.
- Dificultad en el diseño de pruebas si las especificaciones no están bien definidas.

Pruebas de caja blanca:

► **Definición:** evaluación del *software* con conocimiento completo de su estructura interna.

► **Ventajas:**

- Permite una cobertura exhaustiva de todas las rutas de ejecución.
- Detecta errores en las fases tempranas del desarrollo.
- Facilita la optimización del código.

► **Desventajas:**

- Requiere conocimientos técnicos profundos.
- Los *scripts* de prueba pueden necesitar ajustes frecuentes si el código cambia regularmente.

Entrenamiento 2

► Planteamiento del ejercicio

Compara las pruebas funcionales y las pruebas de rendimiento.

► Desarrollo paso a paso

1. Define qué son las pruebas funcionales y las pruebas de rendimiento.

2. Identifica las principales diferencias entre ambas.

3. Explica las ventajas y desventajas de cada tipo de prueba.

► Solución

Pruebas funcionales:

► **Definición:** evaluación del *software* para verificar que cumple con los requisitos funcionales definidos.

► **Ventajas:**

- Asegura que el *software* responde correctamente a las entradas proporcionadas.
- No requiere conocimiento de la estructura interna del *software.*
- Detecta errores en la funcionalidad general del sistema.

► **Desventajas:**

- No evalúa el rendimiento del *software* bajo condiciones de carga.
- Puede no detectar problemas de seguridad o de rendimiento.

Pruebas de rendimiento:

- **Definición:** evaluación del *software* para medir su capacidad de manejar grandes volúmenes de datos y usuarios sin degradar su desempeño.
- **Ventajas:**
    - Identifica cuellos de botella y problemas de rendimiento.
    - Asegura que el *software* puede manejar la carga esperada en producción.
    - Evalúa la robustez y estabilidad del sistema bajo condiciones extremas.
- **Desventajas:**
    - Requiere entornos de prueba que simulen fielmente el entorno de producción.
    - Puede ser costoso y complejo de implementar.

Entrenamiento 3

- Planteamiento del ejercicio

Realiza una prueba unitaria en JUnit5 para una función que suma dos números.

- Desarrollo paso a paso

1. Crea una función sumar(*a*, *b*) que devuelva la suma de *a* y *b.*
2. Utiliza JUnit para escribir una prueba unitaria que verifique que sumar(2, 3) devuelve 5.
3. Ejecuta la prueba y verifica los resultados.

- Solución

```
public class SumaTest {
    @Test
    public void testSumar() {
        assertEquals(5, sumar(2, 3));
    }

    public int sumar(int a, int b) {
        return a + b;
    }
```

```
}
```

Entrenamiento 4

► Planteamiento del ejercicio

Realiza en JUnit5 una prueba de caja negra para una función que calcula el factorial de un número.

► Desarrollo paso a paso

1. Implementa una función factorial(*n*) que calcule el factorial de *n.*
2. Escribe casos de prueba que verifiquen el comportamiento de la función para diferentes valores de *n* sin conocer la implementación interna.
3. Ejecuta las pruebas y verifica los resultados.

► Solución

```
public class FactorialTest {
    @Test
    public void testFactorial() {
        assertEquals(120, factorial(5));
        assertEquals(1, factorial(0));
        assertEquals(1, factorial(1));
    }

    public int factorial(int n) {
        if (n == 0) return 1;
        return n * factorial(n - 1);
    }
}
```

Solución con parametrización

```
public class FactorialTest {
    @ParameterizedTest
    @CsvSource({
        "5, 120",
```

```
        "0, 1",
        "1, 1"
    })
    void testFactorial(int input, int expected) {
        assertEquals(expected, factorial(input));
    }

    public int factorial(int n) {
        if (n == 0) return 1;
        return n * factorial(n - 1);
    }
}
```

Entrenamiento 5

► Planteamiento del ejercicio

Genera una tabla de aserciones de JUnit con la descripción de cada una.

► Desarrollo paso a paso

1. Revisa las aserciones de JUnit mencionadas en el documento.
2. Crea una tabla que incluya el nombre de cada aserción y una breve descripción de su propósito.

► Solución

Aserción	Descripción
assertArrayEquals	Verifica que dos *arrays* sean iguales en contenido y orden.
assertEquals	Comprueba que dos valores sean iguales.
assertTrue	Asegura que una condición sea verdadera.
assertFalse	Asegura que una condición sea falsa.
assertNull	Verifica que un objeto sea *null*.
assertNotNull	Verifica que un objeto no sea *null*.
assertSame	Verifica que dos referencias apunten al mismo objeto.
assertNotSame	Verifica que dos referencias no apunten al mismo objeto.
assertAll	Permite agrupar varias aserciones en una sola llamada.

Tabla 2. Solución. Fuente: elaboración propia.

Test

1. ¿Cuál es el propósito principal de las pruebas funcionales?

A. Evaluar la estructura interna del código.

B. Verificar la interacción entre diferentes módulos del *software.*

_ C. Comprobar que el *software* cumple con los requisitos funcionales definidos.

D. Medir el rendimiento del *software* bajo condiciones de carga.

La opción C es correcta porque las pruebas funcionales se enfocan en verificar si el sistema responde adecuadamente a las entradas proporcionadas según las especificaciones funcionales.

2. ¿Qué tipo de prueba se realiza sin ejecutar el código del *software*?

A. Pruebas dinámicas.

_ B. Pruebas estáticas.

C. Pruebas de rendimiento.

D. Pruebas de integración.

La opción B es correcta porque las pruebas estáticas se realizan sin necesidad de ejecutar el código, centrándose en la revisión de documentos, diseño y código fuente.

3. ¿Cuál de las siguientes afirmaciones describe mejor las pruebas de caja negra?

_ A. Se centran en la funcionalidad del sistema sin conocer su estructura interna.

B. Se basan en el conocimiento completo del código fuente.

C. Se realizan durante la fase de desarrollo del *software.*

D. Se enfocan en la interacción entre diferentes unidades de código.

La opción A es correcta, porque las pruebas de caja negra examinan la funcionalidad del *software* sin tener en cuenta cómo está implementado internamente.

4. ¿Qué entorno se utiliza para realizar pruebas antes de lanzar el *software* a producción?

A. Entorno de desarrollo.

_ B. Entorno de preproducción.

C. Entorno de integración.

D. Entorno de pruebas de sistema.

La opción B es correcta, porque el entorno de preproducción simula el entorno de producción para realizar pruebas exhaustivas antes del lanzamiento.

5. ¿Qué tipo de pruebas se enfoca en evaluar la robustez y estabilidad del *software* bajo condiciones extremas?

A. Pruebas de carga.

_ B. Pruebas de estrés.

C. Pruebas funcionales.

D. Pruebas de usabilidad.

La opción B es correcta, porque las pruebas de estrés evalúan cómo el sistema se comporta bajo situaciones extremas, como sobrecarga o fallos.

6. ¿Qué objetivo tiene la validación en el proceso de pruebas de *software*?

A. Asegurar que el *software* no tiene errores.

_ B. Confirmar que el *software* cumple con las necesidades reales del usuario.

C. Verificar que el *software* cumple con las especificaciones técnicas.

D. Evaluar el rendimiento del *software.*

La opción B es correcta, porque la validación se centra en garantizar que el *software* satisface las necesidades y expectativas del cliente.

7. ¿Cuál es la principal ventaja de las pruebas unitarias?

_ A. Detectar errores en las unidades más pequeñas del código antes de la integración.

B. Evaluar el rendimiento del *software* en su conjunto.

C. Medir la usabilidad del *software.*

D. Verificar la seguridad del sistema.

La opción A es correcta, porque las pruebas unitarias se enfocan en asegurar que las unidades individuales de código funcionen correctamente antes de integrarlas.

8. ¿Qué pruebas se realizan para verificar que un sistema se recupera adecuadamente tras un fallo?

A. Pruebas de carga.

B. Pruebas de seguridad.

C. Pruebas de rendimiento.

_ D. Pruebas de recuperación.

La opción D es correcta, porque las pruebas de recuperación evalúan la capacidad del sistema para recuperarse después de un fallo.

9. ¿Cuál es el enfoque principal de las pruebas de usabilidad?

A. Verificar la seguridad del sistema.

_ B. Evaluar la facilidad de uso de la aplicación.

C. Medir el rendimiento bajo condiciones de alta carga.

D. Asegurar que el *software* cumple con las especificaciones técnicas.

La opción B es correcta, porque las pruebas de usabilidad se centran en evaluar qué tan fácilmente los usuarios pueden interactuar con la aplicación.

10. ¿Qué tipo de pruebas se utilizan para verificar que los cambios en el *software* no han introducido nuevos errores?

_ A. Pruebas de regresión.

B. Pruebas de integración.

C. Pruebas de sistema.

D. Pruebas de validación.

La opción A es correcta, porque las pruebas de regresión se realizan para asegurarse de que las modificaciones en el *software* no han afectado negativamente su funcionamiento.

11. ¿Cuál es la principal característica de las pruebas estructurales?

A. No requieren la ejecución del código.

B. Se centran en la funcionalidad del *software.*

_ C. Evalúan el código con un conocimiento completo de su estructura interna.

D. Son realizadas por los usuarios finales.

La opción C es correcta, porque las pruebas estructurales, también conocidas como pruebas de caja blanca, analizan el funcionamiento interno del *software* basándose en el código.

12. ¿Cuál es el propósito de las pruebas de integración?

A. Verificar que las unidades individuales de código funcionan correctamente.

_ B. Evaluar la interacción entre diferentes componentes del sistema.

C. Medir la accesibilidad del *software.*

D. Probar el rendimiento del *software* bajo carga.

La opción B es correcta, porque las pruebas de integración se centran en comprobar que los diferentes módulos del sistema interactúan de manera adecuada.

13. ¿Qué tipo de prueba está destinada a asegurar que el *software* es accesible para usuarios con discapacidades?

A. Pruebas de rendimiento.

_ B. Pruebas de accesibilidad.

C. Pruebas de integración.

D. Pruebas de caja negra.

La opción B es correcta, porque las pruebas de accesibilidad se enfocan en garantizar que el *software* puede ser utilizado por personas con diferentes tipos de discapacidades.

14. ¿Qué tipo de prueba se realiza en un entorno que replica lo más fielmente posible el entorno de producción?

A. Pruebas unitarias.

_ B. Pruebas de preproducción.

C. Pruebas de caja blanca.

D. Pruebas de usabilidad.

La opción B es correcta, porque las pruebas de preproducción se realizan en un entorno que simula el entorno de producción para verificar que el *software* funcionará correctamente cuando sea lanzado.

15. ¿Cuál es la principal desventaja de las pruebas de caja negra?

A. Requieren conocimientos técnicos profundos.

_ B. Pueden no cubrir todas las rutas de ejecución del código.

C. No se centran en la funcionalidad del *software.*

D. Son difíciles de automatizar.

La opción B es correcta, porque las pruebas de caja negra se basan en un conjunto limitado de entradas y pueden no explorar todas las posibles rutas de ejecución en el código.

16. ¿Cuál es el objetivo principal de las pruebas de rendimiento?

_ A. Evaluar cómo se comporta el *software* bajo condiciones de carga.

B. Verificar la accesibilidad del *software.*

C. Probar la interacción entre módulos de *software.*

D. Asegurar que el *software* cumple con los requisitos técnicos.

La opción A es correcta, porque las pruebas de rendimiento se centran en medir la capacidad del *software* para manejar grandes volúmenes de datos y usuarios sin degradar su desempeño.

17. ¿Qué tipo de pruebas se utilizan para verificar si el sistema protege adecuadamente contra accesos no autorizados?

A. Pruebas de recuperación.

_ B. Pruebas de seguridad.

C. Pruebas de carga.

D. Pruebas de usabilidad.

La opción B es correcta, porque las pruebas de seguridad están diseñadas para evaluar la capacidad del sistema de protegerse contra accesos indebidos o amenazas externas.

18. ¿Qué herramientas se utilizan para automatizar pruebas unitarias en aplicaciones Java?

A. Selenium.

B. Cucumber.

_ C. JUnit.

D. Appium.

La opción C es correcta, porque JUnit es una herramienta ampliamente utilizada para la automatización de pruebas unitarias en aplicaciones Java.

19. ¿Qué técnica de prueba permite probar el *software* desde la perspectiva del usuario sin conocer la implementación interna?

_ A. Pruebas de caja negra.

B. Pruebas de caja blanca.

C. Pruebas de integración.

D. Pruebas de rendimiento.

La opción A es correcta, porque las pruebas de caja negra permiten evaluar el comportamiento del *software* basándose únicamente en sus especificaciones y entradas, sin necesidad de conocer su código fuente.

20. ¿Cuál es el propósito de un plan de pruebas?

A. Probar solo las unidades de código individuales.

B. Crear un entorno de desarrollo.

_ C. Planificar y organizar las pruebas para garantizar una cobertura completa del *software.*

D. Medir la velocidad del código fuente.

La opción C es correcta. porque un plan de pruebas es crucial para estructurar y guiar el proceso de pruebas de *software,* asegurando que todos los aspectos importantes del sistema sean evaluados.

21. ¿Qué tipo de prueba asegura que un sistema no solo cumple con las especificaciones técnicas, sino que también satisface las necesidades del usuario final?

A. Pruebas de integración.

B. Pruebas estructurales.

_ C. Pruebas de validación.

D. Pruebas unitarias.

La opción C es correcta, porque las pruebas de validación están orientadas a confirmar que el *software* cumple con las necesidades y expectativas del usuario final.

22. ¿Qué tipo de prueba se realiza para verificar el funcionamiento de rutas específicas dentro de una unidad de código?

_ A. Pruebas unitarias.

B. Pruebas de sistema.

C. Pruebas de regresión.

D. Pruebas de caja negra.

La opción A es correcta, porque las pruebas unitarias se centran en verificar rutas específicas dentro de las unidades de código más pequeñas, como funciones o métodos, asegurando su correcto funcionamiento.

23. ¿Cuál es la función principal de JUnit en el desarrollo de *software*?

A. Probar la usabilidad del *software.*

_ B. Automatizar las pruebas unitarias en aplicaciones Java.

C. Verificar la accesibilidad del *software.*

D. Realizar pruebas de rendimiento bajo carga.

La opción B es correcta, porque JUnit es un marco de trabajo utilizado específicamente para la automatización de pruebas unitarias en aplicaciones Java, permitiendo a los desarrolladores asegurar la calidad del código.

24. ¿Cuál de las siguientes anotaciones en JUnit se utiliza para definir un método que debe ejecutarse antes de cada método de prueba?

_ A. @Before.

B. @Test.

C. @After.

D. @RunWith.

La opción A es correcta, porque @Before es la anotación en JUnit que indica que un método debe ejecutarse antes de cada método de prueba, preparándolo para su ejecución.

25. ¿Cuál de los siguientes métodos de aserción de JUnit verifica que dos valores son iguales?

A. `assertTrue()`.

_ B. `assertEquals()`.

C. `assertNull()`.

D. `assertSame()`.

La opción B es correcta, porque el método `assertEquals()` en JUnit se utiliza para comparar dos valores y verificar que son iguales, marcando la prueba como fallida si no lo son.

26. ¿Qué anotación de JUnit se utiliza para agrupar y ejecutar conjuntamente varias clases de prueba?

A. @Test.

B. @Before.

C. @After.

_ D. @RunWith.

La opción D es correcta, porque @RunWith se utiliza en JUnit para especificar un ejecutor de pruebas personalizado y para agrupar y ejecutar múltiples clases de prueba como parte de una suite de pruebas.

27. ¿Qué método de aserción de JUnit se utiliza para verificar que un objeto no es null?

A. `assertNull()`.

B. `assertEquals()`.

_ C. `assertNotNull()`.

D. `assertTrue()`.

La opción C es correcta, porque `assertNotNull()` se usa en JUnit para comprobar que un objeto no es *null,* asegurando que una instancia de objeto ha sido correctamente inicializada.