



Guía QA

QUALITY ASSURANCE

Versión 1.0

izertis

Contenido

Introducción	4
Fundamentos del QA Testing	6
1.1 Entendiendo el proceso del SQA.....	6
1.2 Objetivos del SQA	6
1.3 QA & QC en el proceso de calidad del software.....	8
1.4 Estándares de calidad del software	10
1.5 Los siete (7) principios fundamentales	12
1.6 Diferencias entre errores, defectos y fallos	14
1.7 Roles en equipos de QA y habilidades esenciales.....	16
1.8 Los beneficios de implementar QA Testing.....	28
El QA Testing en el Ciclo de Vida del Desarrollo de Software (SDLC)	29
2.1 El QA testing en el contexto SDLC	29
2.2 Pruebas en las distintas metodologías de desarrollo de software.....	30
2.3 Niveles de prueba.....	44
2.4 Tipos de pruebas	45
Técnicas de diseño de casos de pruebas	48
3.1 Técnica de diseño de pruebas basada en la experiencia.....	48
3.2 Técnica de diseño de pruebas de caja blanca	52
3.3 Técnica de diseño de pruebas de caja negra	55
3.4 Especificación de casos de prueba	58
Gestión de actividades de prueba	73
4.1 Estrategia y planificación de pruebas.....	73
4.2 Monitorización y control de pruebas.....	74
4.3 Análisis de pruebas	76
4.4 Diseño de pruebas	77
4.5 Implementación de pruebas	78
4.6 Ejecución de pruebas	78
4.7 Cierre de pruebas	79
Herramientas de soporte para el proceso de QA Testing	81
5.1 Herramientas de gestión de pruebas.....	81
5.2 Herramientas de automatización de pruebas	83
5.3 Herramientas de monitoreo y análisis	88

5.4 Herramientas de performance	90
5.5 Herramientas de calidad de código	92
5.6 Herramientas de seguridad	93
5.7 Herramientas para evaluar accesibilidad web	95
Métricas y KPIs esenciales en QA	98
6.1 Métricas vs KPIs: enfoque y propósito	98
6.2 KPIs fundamentales.....	101
6.3 Quality Gates o rangos de calidad	104
Buenas prácticas para un QA Testing exitoso	105
7.1 Estrategia QA a la medida	105
7.2 Diseño integral de planes de prueba.....	107
7.3 Selección de herramientas adecuadas	109
7.4 Estrategia CI & CD.....	109
7.5 Integración de BDD y TDD en el proceso de desarrollo Agile	111
7.6 Combinación de pruebas manuales y automatizadas	112
7.7 Comunicación y trabajo en equipo.....	112
7.8 Reporte y monitoreo de resultados.....	112



Introducción

Esta guía de QA te proporcionará un conocimiento amplio de todos los conceptos, técnicas, procedimientos y herramientas para que puedas comenzar a trabajar en la mejora de la calidad de tu proyecto software, dividido en los siguientes capítulos:

1

Fundamentos del QA Testing

Comenzamos con una visión general del **proceso de SQA** (Aseguramiento de la Calidad de Software), sus **objetivos**, y las diferencias entre **QA** y **QC** (Control de Calidad). Aprenderás los **principios fundamentales** que sustentan cualquier estrategia de calidad y cómo identificar **errores, defectos y fallos**. Además, se abordan los roles clave en los equipos de QA y sus habilidades esenciales para garantizar un flujo de trabajo efectivo.

2

El QA Testing en el Ciclo de Vida del Desarrollo de Software (SDLC)

Este capítulo explora cómo se integra el QA en cada fase del **SDLC**, desde el análisis hasta el mantenimiento. Descubre cómo aplicar pruebas de calidad en diferentes **metodologías de desarrollo** y los **tipos de pruebas** que debes implementar en cada etapa.

3

Técnicas de QA Testing

Aquí profundizamos en diversas **técnicas de diseño de pruebas**, como las basadas en la **experiencia, caja blanca y caja negra**. Te enseñamos cómo diseñar casos de prueba efectivos para maximizar la cobertura y detectar errores en los momentos más críticos del ciclo de desarrollo.

4

Gestión de actividades de prueba

El capítulo aborda cómo **planificar y gestionar** las pruebas de manera efectiva, desde la **estrategia de pruebas** hasta el **análisis y monitoreo** de resultados. También cubre cómo **ejecutar pruebas** y llevar a cabo un **cierre de pruebas** adecuado para asegurar que el proceso sea eficiente y transparente.

5

Herramientas de soporte para QA Testing

En esta sección te mostramos las mejores **herramientas** disponibles para optimizar el proceso de QA, desde la **gestión de pruebas** hasta la **automatización**. Incluye herramientas para **monitoreo, análisis de rendimiento, calidad de código**, y más.

6

Métricas y KPIs esenciales en QA

Aquí descubrirás cómo establecer **métricas y KPIs** clave para medir el éxito de tus esfuerzos en QA. Aprenderás sobre **Quality Gates** y cómo asegurarte de que el proceso de calidad sea alineado con los objetivos del proyecto.

7

Buenas prácticas para un QA Testing exitoso

Finalmente, esta sección ofrece una recopilación de **estrategias probadas** para garantizar el éxito de QA, como el diseño de **planes de prueba integrales**, la selección de **herramientas adecuadas** y la integración de enfoques como **CI/CD, BDD y TDD** en un entorno **Agile**.

1

Fundamentos del QA Testing

1.1 Entendiendo el proceso del SQA

En el proceso de QA Testing es esencial aplicar el aseguramiento de la calidad software (por sus siglas en inglés, SQA: Software Quality Assurance) en los proyectos de desarrollo y mantenimiento del software, como se podrá deducir de la lectura de esta guía. No se limita a una simple ejecución de pruebas y verificación de resultados, sino que involucra una serie de actividades, técnicas y herramientas complementarias que contribuyen a la detección de defectos en el software, permiten validar el cumplimiento de los requerimientos del software en coherencia con las expectativas de las partes interesadas y/o usuarios finales, su propósito es evaluar la calidad del software y reducir el riesgo de fallos en su funcionamiento, además, puede realizarse en diferentes etapas del ciclo de vida del software y su importancia varía según la naturaleza y requerimientos del proyecto.

1.2 Objetivos del SQA

El SQA tiene como finalidad asegurar que el software desarrollado cumpla con altos estándares de calidad y expectativas de los usuarios. A continuación, enumeramos algunos objetivos fundamentales del SQA:

1. Prevención de defectos: El objetivo principal es prevenir la aparición de defectos en el software, a través de una estrategia y procesos bien definidos en todas las fases del ciclo de desarrollo del proyecto.

2. Mejora continua de procesos: Identificar oportunidades de mejora continua de los procesos de desarrollo y pruebas, evaluando y ajustando la metodología de trabajo actual del equipo, así como también, fomentar la adopción de nuevas prácticas.

3. Cumplimiento de estándares: Garantizar que el software se desarrolle conforme a los estándares de calidad, regulaciones y directrices de la industria y la organización.

4. Planificación y gestión de la calidad: Definir los planes de prueba con objetivos, métricas y procesos de evaluación, que aseguren el control de calidad a lo largo del ciclo de vida de desarrollo software en el proyecto.

5. Reducción de riesgos: Identificar y abordar proactivamente los riesgos asociados con el desarrollo de software, incluyendo riesgos técnicos, operativos y de cumplimiento.

6. Control de cambios: Gestionar adecuadamente los cambios realizados en el software, asegurándose de evaluar tanto el impacto como las validaciones necesarias de manera correspondiente.

7. Documentación y trazabilidad: Realizar una documentación detallada de todos los procesos y decisiones durante el desarrollo, del proyecto, para brindar seguimiento y garantizar la responsabilidad. Por otro lado, la trazabilidad relaciona cada definición de la funcionalidad del sistema con los casos de uso, casos de prueba, ejecuciones, resultados, defectos y su resolución, etc, es imprescindible para conocer la calidad de nuestro sistema, calcular la cobertura que tiene cada requisito por los tests, etc.

8. Involucramiento continuo del equipo de QA: El equipo de QA debe participar activamente en todas las etapas SDLC (por sus siglas en inglés: Software Development Live Cycle) del proyecto, desde la planificación hasta la entrega del software, asegurando el cumplimiento de los estándares de calidad.

9. Evaluación del rendimiento: Evaluar el rendimiento del software mediante pruebas, revisiones y mediciones periódicas para asegurar el cumplimiento de las métricas de rendimiento esperadas.

10. Satisfacción del cliente: El objetivo final es satisfacer las necesidades y expectativas de los clientes, manteniendo su confianza y satisfacción.

1.3 QA & QC en el proceso de calidad del software

El Aseguramiento de Calidad (por sus siglas en inglés QA: Quality Assurance) y el Control de Calidad (por sus siglas en inglés QC: Quality Control) son dos conceptos esenciales relacionados en la calidad del desarrollo de software, pero se enfocan en diferentes aspectos del proceso. A continuación, listamos las principales características que los diferencian:

Aseguramiento de la calidad (QA)

Preventivo

Es un enfoque más preventivo para gestionar la calidad, centrado en mejorar la definición de los procesos, estándares y procedimientos para prevenir defectos desde el inicio del desarrollo.

Actividades de planificación

Se incluyen actividades claves para elaborar planes de calidad, establecer los objetivos, las estrategias, los recursos y los procesos para el control de calidad durante todo el SDLC del proyecto.

Orientado a procesos

Fundamentalmente se centra en la calidad de los procesos utilizados para desarrollar el software, para asegurar una correcta definición, seguimiento y gestión.

Prevención de defectos

Su propósito es prevenir los defectos mejorando los procesos y adoptando buenas prácticas de desarrollo, por lo que reduce la probabilidad de encontrar defectos en el software.

Involucramiento continuo

Está implicado en todo SDLC del proyecto de software, desde la planificación hasta la entrega, asegurando el cumplimiento de los estándares de calidad en cada fase del proyecto.

Control de calidad (QC)

Enfoque retrospectivo

Se centra en verificar productos de software para identificar defectos y problemas de calidad después de su desarrollo.

Actividades de verificación y validación

Incluye actividades como pruebas, revisiones, inspecciones y verificación de documentos para detectar defectos en el software y sus componentes.

Orientado al software

Evalúa la calidad del software en sí, asegurando que cumpla con los estándares de calidad establecidos (sin enfocarse necesariamente en los procesos de desarrollo).





Corrección de defectos

Su propósito principal es identificar y corregir defectos o reducirlos para ofrecer un software de alta calidad.

Involucramiento en etapas específicas

Se centra en la etapa de prueba y validación, que ocurre después del desarrollo de software, siendo en una fase específica en el SDLC del proyecto.

Hemos visto los aspectos fundamentales del QA y QC en la calidad del desarrollo de Software, en resumen, podemos concluir:

	Aseguramiento de la calidad (QA)	Control de calidad (QC)
 Propósito	Asegurar la calidad del proceso de desarrollo de software.	Controlar la calidad del software.
 Alcance	Se aplica en todo el SDLC (desde la planificación hasta la implementación).	Ejecuta principalmente en la fase de pruebas.
 Enfoque	Preventivo, orientado al proceso, centrado en establecer estándares, procesos y procedimientos para garantizar la calidad en todo el SDLC.	Correctivo, orientado al producto se centra en ejecutar las pruebas para encontrar defectos en el software.
 Resultado	Garantiza que se cumplan los estándares de calidad en todo el SDLC.	Identifica los defectos en el software para que puedan ser corregidos antes de su lanzamiento.

Es fundamental entender que las pruebas son una parte del control de calidad, el cual a su vez es una parte del aseguramiento de calidad. El QA tiene un alcance más amplio, centrado en la implementación de procesos. Por otro lado, el QC se ocupa de los aspectos de ejecución y verificación, incluyendo las pruebas y otras.



1.4 Estándares de calidad del software

Actualmente, son varias las organizaciones internacionales que establecen y unifican buenas prácticas y estándares entorno a la calidad del desarrollo, entre ellas:



Organización Internacional de Normalización (International Organization for Standardization)

Sus normativas especifican requerimientos para garantizar que los productos y/o servicios cumplen con su objetivo.



Instituto de Ingenieros en Electricidad y Electrónica (Institute of Electrical and Electronic Engineers)

Sus normativas tienen como fin unificar la forma de presentar trabajos escritos a nivel internacional.



Comisión Electrotécnica Internacional (International Electrotechnical Commission)

Sus normativas son documentos técnicos que ayudan a diseñadores y fabricantes a garantizar la seguridad.



Una Norma Española

Sus normativas se crean en los Comités Técnicos de Normalización (CTN) de la Asociación Española de Normalización y Certificación (AENOR) e incluyen adaptaciones españolas de normas internacionales.



A continuación, nos centraremos en la normativas o estándares de calidad más relevantes en la industria del desarrollo de software:

- **ISO 9001:** es una normativa para la implementación de un método o un SGC (Sistema de Gestión de la Calidad), acreditando de la capacidad para cumplir los requisitos de calidad. La norma establece requisitos aplicables a cualquier organización.
- **ISO 10005:2018:** proporciona una guía para gestionar un plan de calidad a lo largo de todo el ciclo de vida de un producto.
- **ISO/IEC/IEE 29119:** normativa para la documentación de prueba de software. Standard for Software Test Documentation (Estándar de Documentación de Pruebas de Software), este estándar reemplaza a IEE 829 – 1998, se centra en la relación de las pruebas con las metodologías de desarrollo y el ciclo de vida software. Describe el papel de las pruebas en la gestión de la calidad y como parte de la verificación y validación del software.
- **ISO/IEC 25000:** supone una familia de normas también conocida como SQuaRE (System and software Quality Requirements and Evaluation), define un marco para evaluar la calidad de los productos de software en áreas como adecuación funcional, fiabilidad, eficiencia, usabilidad, seguridad, compatibilidad, portabilidad y mantenibilidad.
- **ISO 3300:** se enfoca en la evaluación de la calidad de los procesos de desarrollo de software.
- **CMMI (por sus siglas en inglés Capability Maturity Model Integration):** provee un marco de referencia para evaluar y mejorar la madurez de los procesos en el desarrollo de software.
- **ISO 12207:** proporciona un estándar para observar los procesos de ciclo de vida del software, desde la idea inicial hasta la retirada del software.
- **IEEE 730 – 2002:** es un estándar para la elaboración de un Plan de Aseguramiento de la calidad de software (por sus siglas en inglés, SQAP, Software Quality Assurance Plan) que define las directrices para asegurar un software de alta calidad y propone una.
- **ISO 5055 Software Quality Standards:** normativas orientadas a medir las debilidades críticas de un software, haciendo énfasis en criterios de seguridad, confianza, eficiencia de rendimiento y mantenibilidad.

El ISTQB (International Software Testing Qualifications Board, por sus siglas en inglés), es una organización de certificación internacional, que utiliza estos entre otros estándares para unificar buenas prácticas y normativas en el ámbito de las pruebas y la calidad del software.

1.5 Los siete (7) principios fundamentales

Hoy en día, muchos profesionales del mundo del SQA suelen adoptar diversas estrategias y enfoques de calidad, es por ello, que, al margen de las técnicas o métodos de prueba, es fundamental conocer los siete (7) principios básicos de las pruebas establecidos en el estándar ISTQB. Estos principios actúan como directrices universales aplicables a todas las pruebas de software, lo cual nos permite ser eficientes en el proceso de pruebas y asegurar la calidad del desarrollo.

- 1 Las pruebas muestran la presencia de defectos.** Las pruebas pueden demostrar la existencia de errores, no su ausencia. Es fundamental partir de esta base y tener claridad sobre el objetivo de las pruebas: identificar y corregir errores, para reducir la cantidad de defectos no descubiertos, lo que minimiza el riesgo de un funcionamiento incorrecto en las operaciones del sistema.
- 2 Las pruebas exhaustivas son imposibles.** Existen muchos tipos de prueba, por lo que llevar a cabo pruebas exhaustivas para validar todas las combinaciones de entradas, precondiciones y postcondiciones, no es posible, excepto en casos triviales o productos digitales básicos. Se recomienda realizar un análisis previo para delimitar qué probar y cómo probar, la estrategia debe priorizar la ejecución de las pruebas más importantes, con el objetivo de obtener la mejor cobertura posible en las funcionalidades más críticas e importantes.
- 3 Las pruebas tempranas ahorran tiempo y dinero.** Encontrar defectos en etapas tempranas del ciclo de desarrollo, permite reducir costos en tiempo y dinero. Este principio es crucial para cualquier proyecto de software, y la razón principal por la que se deben iniciar las pruebas desde la fase del desarrollo, cuanto más tarde se encuentren los defectos, más costoso es solucionarlos.
- 4 Agrupación de defectos.** A menudo, la mayoría de los defectos tienden a concentrarse en un componente o funcionalidad, generalmente, por realizar cambios o evolutivos en el componente o funcionalidad, o por arreglar algún defecto, se puede introducir otro. Se deben identificar y/o agrupar estos defectos en aras de aplicar los correctivos con mayor efectividad.

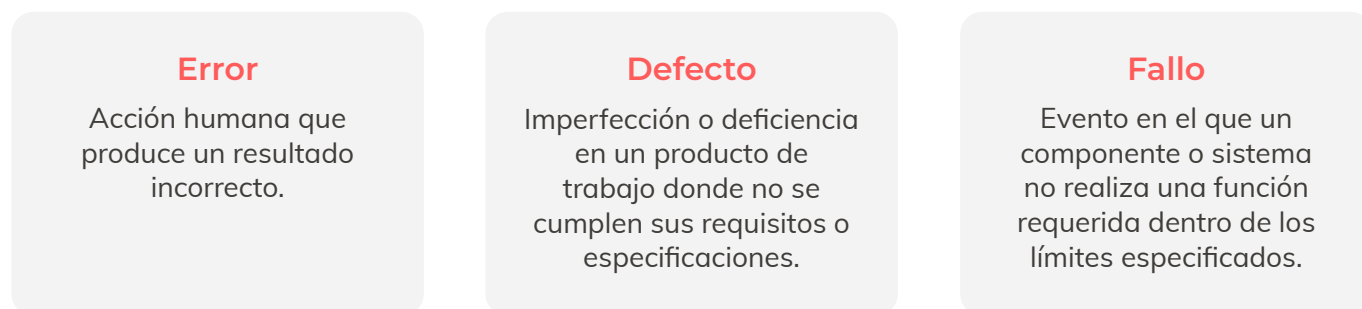
- 5 Paradoja del pesticida.** Si se repiten siempre las mismas pruebas (sin cambios), finalmente no se detectarán nuevos defectos, por lo que algunos casos pueden perder su efectividad, por eso se recomienda mantener actualizada la batería de pruebas, incorporando nuevos casos o actualizar los existentes de forma regular además de realizar pruebas exploratorias sin guión sobre la funcionalidad que haya cambiado en el sistema.
- 6 Las pruebas dependen del contexto.** En general, no existe una fórmula o estrategia de pruebas única que se adapte a las necesidades de cualquier proyecto de software, las pruebas dependerán del escenario, los casos de uso, las condiciones del entorno, entre otras variables, por eso es importante adecuar las pruebas según el contexto específico que se desea probar
- 7 La falacia de ausencia de errores.** Este principio destaca que, por más pruebas para asegurar la calidad de un software, nunca estará exento o libre de defectos (en otras palabras, siempre tendrá defectos), se puede alcanzar cierto grado de estabilidad o madurez mediante las pruebas y lograr que los defectos más críticos se corrijan, pero no garantiza que cumpla con todas las expectativas. Se puede medir la calidad por la gestión efectiva de los defectos, pero nunca se alcanza su completa eliminación.



1.6 Diferencias entre errores, defectos y fallos

En el mundo de la calidad software, se utiliza un lenguaje específico con relación a los errores, de forma que tres palabras que pueden pasar por sinónimos en nuestro ámbito tienen importantes diferencias: error, defecto y fallo.

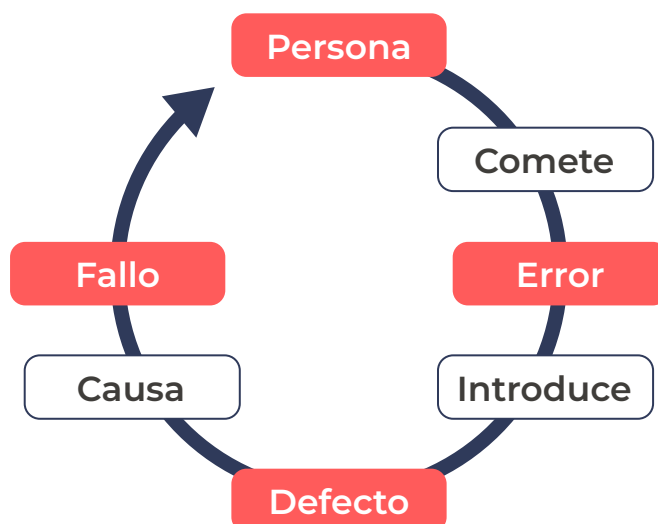
Las definiciones oficiales del glosario de ISTQB en español son:



Centrándonos en el desarrollo de un sistema software podemos concretar más los conceptos:

Error es una equivocación humana (por parte de alguien del equipo de análisis, desarrollo, testing, etc) que produce una incorrección en su trabajo durante el proyecto.

Esto se refleja en un **defecto** (imperfección) en algún artefacto software, por ejemplo: el código fuente de un programa, el documento de especificación de un nuevo requisito, la especificación de un caso de uso o de pruebas... que a su vez puede dar lugar a **fallos** en el programa durante el tiempo de explotación.

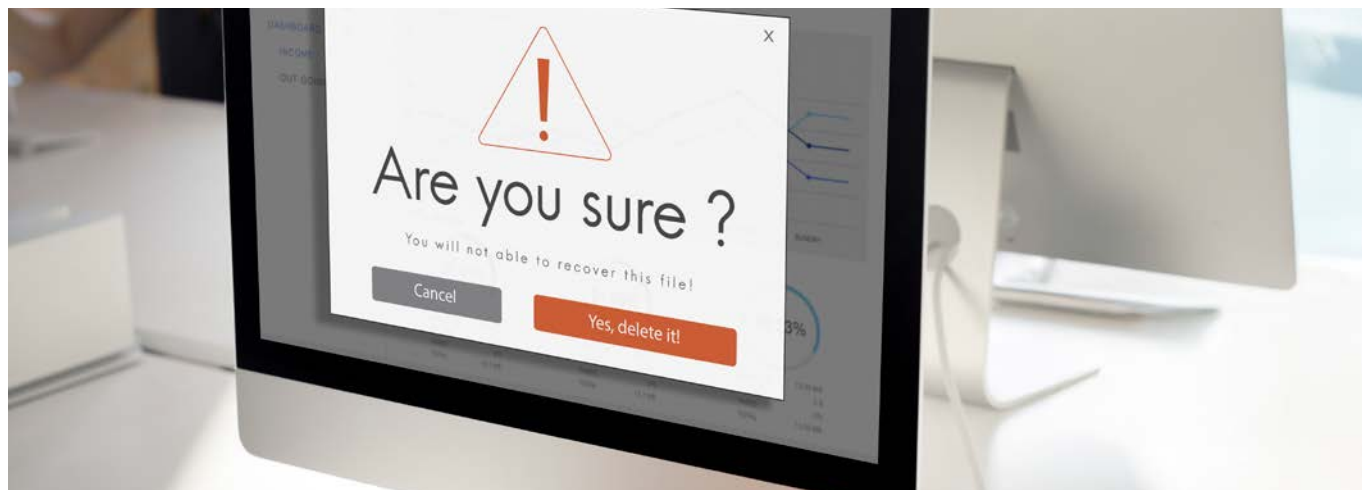


Como vemos un error se puede producirse en cualquier fase y por parte de todos los intervinientes en un proyecto de software. Los seres humanos cometemos errores por distintas razones, como la complejidad del trabajo, la escasez de tiempo o de herramientas adecuadas, los procesos, la comunicación, el cansancio o falta de conocimiento necesario. Aunque errar es humano y, por tanto, no se pueden evitar los errores al 100%, la aplicación de las distintas técnicas de QA ayudarán a prevenir y disminuir los errores.

Los defectos en un artefacto producido en las primeras fases del SDLC si no se encuentran y corrigen, probablemente darán lugar a defectos en artefactos creados más adelante, por ello es importante realizar QA en todas las fases del proyecto. Un defecto en el código de programa puede que no llegue nunca a producir un fallo, si esa pieza de código no se ejecuta o si se ejecuta, pero con unas condiciones diferentes a las requeridas para desencadenar el fallo. Otros defectos de código darán siempre lugar a fallos en el sistema. También existe la posibilidad de que un fallo se produzca no por un defecto sino por condiciones ambientales, como cuando la radiación o el campo electromagnético provocan defectos en el firmware.

Teniendo en cuenta las distinciones anteriores, podemos listar una serie de acciones orientadas a disminuir cada uno de los conceptos:

- Para evitar errores debemos proporcionar a todas las personas implicadas la formación, entrenamiento, entorno de pruebas adecuado, un conjunto de herramientas, técnicas y métodos adaptados al tipo de proyecto.
- Para detectar defectos también podemos apoyarnos en técnicas de análisis, documentación, reuniones “tres amigos”, revisiones por pares, análisis estático y análisis dinámico de código, etc.
- Para encontrar fallos en el sistema tenemos las ejecuciones de los casos de prueba, ya sean previamente definidos o exploratorio.



1.7 Roles en equipos de QA y habilidades esenciales

En el mundo del desarrollo software, hay siempre diversidad de roles entre los que se reparten las tareas y responsabilidades necesarias para completar el proyecto. En esta sección se relacionan algunos de los roles más habituales que trabajan en QA y las habilidades que requieren. Según diversos aspectos como la estructura y la cultura de la organización, la metodología del ciclo de vida de desarrollo software, el tamaño del equipo y del proyecto, etc, se podrán implementar todos o algunos de estos roles, o una variante de ellos.

1.7.1 Gerente de pruebas

Este rol también conocido en el mundo del testing como QA Manager, tiene un enfoque estratégico y se encarga de la planificación a largo plazo y la gestión de recursos a nivel organizacional.



Responsabilidades

Sus responsabilidades incluyen definir la estrategia de QA, coordinar todos los aspectos del proceso de pruebas, gestionar presupuestos y cronogramas, y comunicar resultados a la alta dirección. El o la Gerente de pruebas o QA Manager se asegura de que los procesos y estándares de calidad se mantengan y mejoren continuamente.



Tareas

Las tareas habituales que suele desempeñar una o un gerente de pruebas podrían ser:

- Redacción del documento de estrategia de calidad para una empresa o departamento software.
- Revisión y corrección de los planes de pruebas para los distintos proyectos.
- Selección, definición de planes de formación y carrera para el equipo de QA .
- Análisis de resultados y definición e implementación de acciones correctivas o de mejora en QA a alto nivel.
- Comunicación resultados estratégicos de los procesos de QA a la alta dirección y otros departamentos.



Conocimientos y habilidades esenciales

Quien asuma este rol debe tener una gran variedad de conocimientos, tanto en la parte genérica de gestión como más específica de la calidad, entre otras:

Gestión de equipos

Planificación de proyectos

Gestión de proyectos

Redacción de informes

Frameworks y herramientas de desarrollo y de pruebas (manuales y automatizadas)

Algunas de las habilidades más importantes de este rol:

Capacidad de análisis

Buena comunicación

Liderazgo

Gestión de la presión



Colaboraciones

El desempeño de la gerencia de pruebas o QA Management implica relacionarse con otros roles como:

- El equipo de calidad, sobre todo los niveles de liderazgo y estrategia.
- Gerentes de otras áreas implicadas en el proyecto, como desarrollo, sistemas, recursos humanos, operaciones, etc.
- Dirección de la empresa.
- Clientes.

1.7.2 Líder de pruebas

En un equipo de aseguramiento de calidad (QA), el líder de pruebas, también conocido como QA Lead, lidera el equipo de QA y es responsable de definir la estrategia de pruebas, planificar el proceso de pruebas y coordinar con otros departamentos, como desarrollo y operaciones. En empresas pequeñas puede que no exista el rol de gerente de pruebas, por lo que el líder de pruebas tomará parte de sus responsabilidades que pueden estar compartidas con las gerentes de otras áreas del proyecto, como desarrollo u operaciones. O que no exista ningún(a) líder de pruebas, por lo que todo lo explicado en esta sección se asigna al rol de gerencia.

En empresas grandes, por otro lado, suele implementarse una jerarquía en la que varias personas con el rol de líder de pruebas responden para diferentes proyectos o áreas de la compañía ante la misma gerencia de pruebas. De esta estructura dependerá mucho el perfil del líder, sus tareas, responsabilidades y relaciones.



Responsabilidades

El líder de pruebas desarrolla estrategias de prueba alineadas con los objetivos del proyecto y con la estrategia de pruebas de la empresa, definiendo planes detallados y seleccionando herramientas adecuadas para un proyecto en concreto. Coordina las actividades del equipo de QA, supervisando la ejecución de pruebas, revisando casos de prueba y gestionando defectos.

Además, el líder de pruebas se encarga de la gestión de recursos del equipo, asignando tareas, planificando la capacidad y apoyando al proceso de selección de personal cuando es necesario.

En empresas en que no hay gerente de pruebas, esta persona también se enfoca en la formación y desarrollo profesional de los miembros del equipo. Si existe, tendrá que implementar en su equipo las políticas definidas para la empresa. La colaboración interdepartamental es crucial, trabajando estrechamente con desarrollo, operaciones y producto para integrar las pruebas en todas las etapas del SDLC, facilitando la comunicación y resolución de problemas.

El líder de pruebas realiza auditorías de calidad y revisiones de procesos para identificar áreas de mejora, implementando cambios para optimizar los procesos de QA y mejorar la eficiencia del equipo. También elabora informes detallados sobre el progreso de las pruebas, resultados y métricas clave, presentándolos al o la gerente de pruebas o en su defecto, a la alta dirección y otros participantes para proporcionar una visión clara del estado de calidad del producto y las áreas que requieren atención.



Tareas

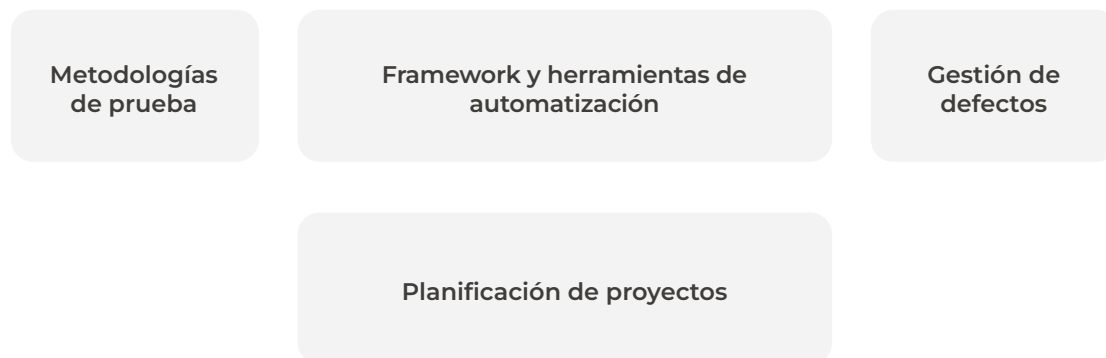
Derivadas de sus responsabilidades, el rol de líder de pruebas verá su día a día ocupado en tareas como:

- Definición de un plan de pruebas para el proyecto acorde a la estrategia general de QA.
- Asignación de tareas a los miembros del equipo de QA.
- Analizar los requisitos del proyecto.
- Planificar las fases de pruebas del proyecto.
- Seguimiento del progreso y resultados de las tareas de QA.
- Implementación de acciones correctivas o de mejora en el proyecto.
- Gestión de defectos.
- Priorización de pruebas y defectos.
- Apoyo al gerente de pruebas en la creación y entrenamiento del equipo de QA.
- Análisis de resultados y definición e implementación de acciones correctivas o de mejora en QA a nivel de proyecto.
- Creación de informes del estado de calidad del proyecto (son cruciales para comunicar los resultados a los participantes y asegurar que todas las partes interesadas tengan una comprensión clara del estado del producto y las áreas que necesitan atención adicional).

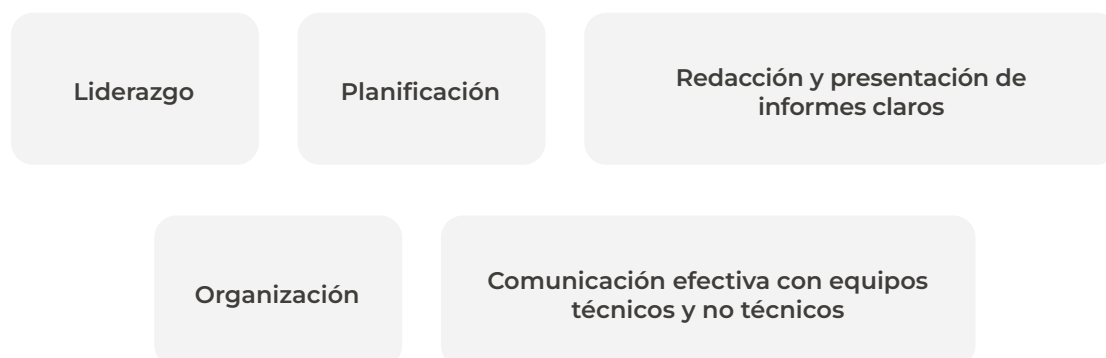


Conocimientos y habilidades esenciales

Un líder de pruebas debe tener conocimientos profundos en:

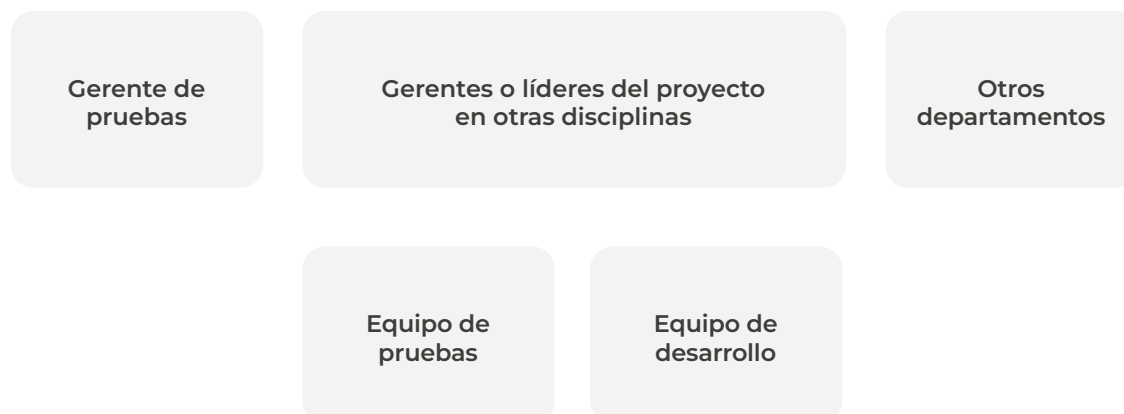


En cuanto a las habilidades, se necesitan:



Colaboraciones

La colaboración interdepartamental es crucial, trabajando estrechamente con desarrollo, operaciones y producto para integrar las pruebas en todas las etapas del ciclo de vida del software, facilitando la comunicación y resolución de problemas.



1.7.3 Ingeniero/a de pruebas

El Ingeniero de pruebas desempeña un papel fundamental en el desarrollo de software al garantizar que los productos sean robustos, funcionales y libres de defectos antes de su lanzamiento. Este rol combina habilidades técnicas con un enfoque meticuloso para diseñar, desarrollar y ejecutar pruebas exhaustivas que evalúen todas las funcionalidades del software.



Responsabilidades

Como se puede deducir del párrafo anterior, sus responsabilidades serán las siguientes:

Realización de las pruebas previstas en el plan de pruebas del proyecto

Reporte y seguimiento de defectos



Tareas

Para cumplir con sus responsabilidades, estas son las tareas más comunes que realiza un ingeniero o ingeniera de pruebas:

- Diseño de casos de prueba detallados que abarcan aspectos como la funcionalidad, rendimiento, seguridad y usabilidad del software. Esto implica tanto pruebas manuales como la automatización de pruebas utilizando frameworks y herramientas adecuadas.
- Identificación y gestión de defectos: utiliza herramientas de seguimiento de errores para documentar y priorizar problemas, colaborando estrechamente con los desarrolladores para reproducir y resolver los defectos de manera efectiva.
- Participa en revisiones de código y diseño para garantizar que se sigan las mejores prácticas de calidad desde las etapas iniciales del desarrollo.
- Integración de pruebas en los procesos de integración y entrega continuas (CI/CD).



Conocimientos y habilidades esenciales

Conocimientos:

Estrategias de definición de casos de prueba

Ejecución de pruebas manuales

Lenguajes de programación de pruebas automáticas

Herramientas de pruebas

Ciclo de vida de los defectos

Habilidades:

Saber trabajar en un entorno ágil es esencial, ya que debe adaptarse rápidamente a los cambios y requerimientos del proyecto

Capacidad analítica y de resolución de problemas



Colaboraciones

Deberá estar en contacto con:

Equipo que desarrolla el programa

Líderes el equipo de QA

Gerencia de QA

Analistas funcionales

1.7.4 Ingeniero/a de automatización

Este rol es muy similar al anterior, pero específico para las pruebas automatizadas. Es un rol fundamental en el campo del aseguramiento de calidad y desarrollo de software, especializado en diseñar, desarrollar y mantener sistemas de pruebas automatizadas. Colabora estrechamente con equipos de desarrollo y QA para identificar oportunidades donde la automatización puede mejorar la eficiencia del proceso de pruebas. Utiliza herramientas y frameworks para crear scripts automatizados que ejecutan pruebas de manera repetitiva y consistente.

La automatización de pruebas no solo mejora la eficiencia al liberar recursos humanos para tareas más estratégicas, sino que también incrementa la cobertura de pruebas al ejecutar casos complejos y extensos de manera repetitiva y confiable. Esto asegura que el software desarrollado cumpla con los estándares de calidad requeridos y permite a los equipos identificar y abordar problemas de manera más efectiva.



Responsabilidades

Su responsabilidad es la automatización de pruebas dentro de un proyecto: este rol no solo se limita a la creación de scripts, sino que también se encarga de su mantenimiento y ejecución, asegurándose de que las pruebas automatizadas estén integradas efectivamente en flujos de integración continua y despliegue continuo CI/CD (por sus siglas en inglés “Continuous integration/Continuous delivery or deployment”).



Tareas

Definición de casos de prueba para automatización

Creación y mantenimiento de pruebas automáticas

Incorporación de estas a los flujos CI/CD del proyecto

Reporte y seguimiento de defectos

Revisión de los resultados



Conocimientos y habilidades esenciales

El Ingeniero de automatización requiere habilidades sólidas en programación y desarrollo de software, junto con experiencia en el diseño y ejecución de pruebas automatizadas. Debe estar familiarizado con múltiples lenguajes de programación y tener un profundo conocimiento de herramientas específicas de automatización.

Conocimientos:

Programación en diversos lenguajes de automatización de pruebas

Diseño de pruebas automáticas

Ejecución de pruebas automáticas

Herramientas de apoyo a la automatización

Habilidades:

Capacidad de análisis

Resolución de problemas

Automotivación

Comunicación

Adaptación a entornos cambiantes



Colaboraciones

Deberá estar en contacto con:

Equipo que desarrolla el programa

Líderes el equipo de QA

Gerencia de QA

Líderes el equipo de QA

1.7.5 Analista de pruebas

Un Analista de pruebas, es un rol usado comúnmente para hacer énfasis en la necesidad de realizar un buen análisis de las funcionalidades del sistema sujeto a pruebas para diseñar y ejecutar correctamente los casos de prueba. Contar con una persona especializada en análisis de pruebas sin duda beneficia al equipo puesto que los casos de prueba serán más exhaustivos, más relevantes y completos.

Aunque existe una certificación concreta de nivel Avanzado de ISTQB, no vamos a profundizar en este rol porque más allá de profundizar en el análisis del Sistema y el diseño metódico de las pruebas, sus responsabilidades, tareas, conocimientos y habilidades y las relaciones son iguales a los dos anteriores.

1.7.6 Consultor QA

Consultor QA es un rol esencial en el aseguramiento de calidad, que se enfoca en mejorar los procesos y prácticas de pruebas dentro de una organización que es cliente de su empresa consultora. Más allá de ejecutar pruebas, este rol implica capacitación, mentoría y asesoría para elevar los estándares de calidad a lo largo del ciclo de desarrollo de software. Puede actuar a nivel de Líder QA, QA Manager o Ingeniero/a QA, dependiendo de las necesidades del cliente y el proyecto al que esté apoyando, pero siempre con el objetivo extra de dar ejemplo y entrenamiento al equipo cliente para que mejore en su área de QA.



Responsabilidades

El consultor de QA desarrolla estrategias de pruebas alineadas con los objetivos del negocio y evalúa las prácticas actuales de QA para proponer mejoras. Define e implementa estándares de calidad y mejores prácticas, asegurando que los equipos de desarrollo y QA usen las herramientas y metodologías más efectivas. Fomentar una cultura de calidad es otra responsabilidad clave. El Consultor QA trabaja para asegurar que todos los miembros del equipo comprendan la importancia del QA y facilita la comunicación y colaboración entre los equipos de desarrollo, QA y operaciones. Promueve la calidad en todas las etapas del ciclo de desarrollo de software, asegurando que el producto final sea de alta calidad y libre de defectos.

Algunas de las tareas que suelen ocupar a un consultor QA de pruebas, dependiendo del nivel de la colaboración con su cliente, son:

- Entrevistas con el cliente para obtener información de su funcionamiento.
- Redacción de documentación de evaluación del estado actual.
- Propuestas de mejoras en el área de QA y organizacional con el objetivo de mejorar la calidad de los procesos y calidad del proyecto del cliente.
- Planificación.
- Ejecución de pruebas manuales y automáticas.
- Impartición de tutoriales o seminarios al equipo del cliente.
- Reporte dirigido al cliente sobre los progresos, los resultados obtenidos y las necesidades identificadas del proyecto.
- Colaboración con otras áreas de consultoría.



Conocimientos y habilidades esenciales

Más allá de los conocimientos específicos de QA, la consultoría requiere talento para tratar con los diferentes contactos en la empresa cliente, podríamos resumir:

Conocimientos:

Pruebas de calidad

Planificación de proyectos

Análisis del estado de madurez de procesos y equipos

Herramientas, frameworks, técnicas, procesos de calidad

Áreas de negocio

Modelos de ciclos de vida de desarrollo

Habilidades:

Comunicación

Coordinación

Liderazgo

Análisis

Adaptación a cambios



Colaboraciones

Una persona trabajando en consultoría QA debe colaborar con con diferentes roles en la empresa cliente:

Equipo de desarrollo

Responsable de proyecto

Dirección

QA

Analistas

Dentro de su propia empresa:

Miembros del equipo de consultoría

Equipo de pruebas manuales

Equipo de pruebas automáticas

Equipo de pruebas no funcionales
(rendimiento, seguridad...)



1.8 Los beneficios de implementar QA Testing

Las pruebas en el desarrollo de productos digitales son fundamentales para asegurar la calidad y el cumplimiento de requisitos funcionales del software, por lo que la incorporación de un enfoque efectivo es una inversión a largo plazo, nos brinda beneficios que contribuyen al éxito del producto y la satisfacción de los clientes.

Entre sus principales beneficios podemos mencionar:



Ahorro de tiempo y reducción de costos.

Las pruebas de software permiten detectar y corregir errores en etapas tempranas de SDLC, lo que evita que los defectos se propaguen y se conviertan en problemas costosos de resolver más adelante. Además, las pruebas ayudan a identificar los errores asociados al diseño para mejorar su eficiencia, lo que también contribuye a reducir los costos de desarrollo y mantenimiento.



Cumplimiento de requisitos.

Las pruebas de aceptación y funcionales garantizan que el desarrollo cumpla con los requisitos del software y las expectativas del mercado.



Mejoras de usabilidad y experiencia del usuario.

Las pruebas funcionales y no funcionales, permiten que el software funcione correctamente en escenarios reales para asegurar que se ajuste a las necesidades de los usuarios, por lo tanto, un aumento de la satisfacción del cliente.



Mayor confiabilidad y mejora de la imagen de la empresa.

Al realizar pruebas exhaustivas, aumenta la confiabilidad y calidad del software, lo que contribuye a satisfacer al cliente y a fortalecer una imagen positiva de la empresa. Por el contrario, un software defectuoso puede dañar la reputación de la empresa y generar una imagen negativa, en consecuencia, la disminución de la cartera de clientes de la empresa.



Reducción de riesgos y mejora de la calidad del software.

Las pruebas permiten identificar y corregir defectos que pueden afectar la seguridad del software o la integridad de la información, lo cual contribuye a proteger la empresa y reducir riesgos asociados a un mal funcionamiento del software.

2

El QA Testing en el Ciclo de Vida del Desarrollo de Software (SDLC)

2.1 El QA testing en el contexto SDLC

El SDLC es un proceso que juega un rol crucial en la producción de software de alta calidad y rentable en el menor tiempo posible. Es un proceso estructurado que separa cada aspecto del desarrollo de software en diferentes fases. Estas fases son:



El QA en el SDLC se refiere a las prácticas de aseguramiento de calidad en las distintas fases de éste, para asegurar que el producto final cumple los estándares de calidad y requerimientos predefinidos. ¿Cómo interviene en las distintas fases?



Análisis

El equipo de QA comienza a revisar y validar los requerimientos del proyecto, asegurando que los mismo sean claros, completos y que estén alineados con los objetivos del proyecto. En esta etapa pueden empezar a definirse las pruebas de aceptación que se realizarán para validar que se han satisfecho todos los requerimientos.



Diseño

Los equipos de QA participan en esta fase para asegurar que la arquitectura del sistema y el diseño son factibles, escalables y alineados con los objetivos de calidad.



Desarrollo

Al trabajar junto a los desarrolladores, el equipo de QA puede realizar revisiones de código para verificar que se aplican los estándares, lanzar pruebas unitarias o ayudar en su diseño. Monitorizar y revisar continuamente el desarrollo ayuda a la detección temprana y solución de errores.



Pruebas

Los equipos de QA ejecutan pruebas funcionales para verificar que el software cumple con los requerimientos funcionales especificados; pruebas no funcionales para verificar que la velocidad, seguridad y otros requisitos no funcionales se cumplan; pruebas de integración para asegurar que las distintas piezas y componentes del software funcionan juntas correctamente; pruebas de regresión para verificar que los nuevos cambios no introduzcan nuevos errores; y pruebas de aceptación de usuario para verificar que el software cumple con sus necesidades y expectativas. Adicionalmente hacen el seguimiento y documentación de las incidencias detectadas.



Despliegue

El equipo de QA verifica que el software se ha desplegado correctamente, y se realizan pruebas en el entorno de producción para comprobar que todo funciona como es debido.



Mantenimiento

Los equipos de QA asisten para verificar que las correcciones de errores o las mejoras en las funcionalidades han sido exitosas. A menudo colaboran en la replicación de los errores encontrados por los usuarios en un entorno de pruebas para que el equipo de desarrollo pueda corregirlos. A través del análisis de estos errores, reciben información que les permite mejorar su librería y priorización de pruebas. Por otra parte, con herramientas de monitoreo, controlan la performance general del sistema y ayudan en la mejora continua tanto del software como del proceso de desarrollo.

2.2 Pruebas en las distintas metodologías de desarrollo de software

El mundo del desarrollo software ha evolucionado en la historia y se han definido distintas metodologías, cada una tiene su propio enfoque y ventajas, dependiendo de cómo responde a las necesidades específicas del proyecto.

Modelo en cascada

El modelo en cascada tiene un enfoque tradicional del desarrollo de software. Se basa en un proceso lineal donde cada fase del desarrollo debe completarse antes de que la siguiente pueda comenzar. Este método es conocido por su estructura clara y fácil de gestionar, pero también ha sido criticado por su rigidez y falta de flexibilidad.



Etapas del modelo en cascada

- **Requisitos:** Recopilación y documentación detallada de todos los requisitos del proyecto.
- **Diseño del sistema:** Creación de la arquitectura del sistema y diseño de componentes.
- **Implementación:** Desarrollo del código basado en el diseño.
- **Pruebas:** Verificación y validación del software a través de pruebas unitarias, de integración, de sistema y de aceptación.
- **Despliegue:** Instalación y puesta en marcha del sistema en el entorno de producción.
- **Mantenimiento:** Corrección de errores y actualizaciones post-despliegue.



Ventajas

- Claridad y estructura bien definidas.
- Documentación exhaustiva.
- Fácil gestión y control del proyecto.



Desventajas

- Rigidez y poca flexibilidad para cambios en los requisitos.
- Detección tardía de errores, lo que puede ser costoso de corregir.
- Retroalimentación de usuarios finales solo al final del ciclo.

Modelo ágil

El modelo ágil se centra en la entrega rápida y continua de pequeñas partes funcionales del software. Es un enfoque iterativo e incremental que fomenta la colaboración y la adaptabilidad. En lugar de seguir un proceso lineal como en la metodología en cascada, el desarrollo ágil se basa en ciclos cortos de trabajo llamados “sprints” o iteraciones, donde se desarrollan y prueban pequeños incrementos de funcionalidad.



Principios del modelo ágil

- **Iteraciones cortas y frecuentes:** Desarrollo en ciclos cortos llamados sprints.
- **Colaboración continua:** Comunicación constante entre desarrolladores, testers y participantes.
- **Adaptabilidad:** Es la capacidad del producto software de ser adaptado a diferentes entornos sin la aplicación de acciones o medios distintos de los aportados para este propósito por el software considerado. [ISO 9126]
- **Pruebas continuas:** Integración de pruebas en cada iteración del desarrollo.



Métodos comunes

Scrum: Scrum es un proceso en el que se aplican de manera regular un conjunto de PRINCIPIOS, PRÁCTICAS y VALORES para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto.

En Scrum se entrega de forma iterativa en sprints o iteraciones incrementos o añadidos de funcionalidad.

- **Roles:** Scrum Master, Desarrollo y Product Owner.
- **Artefactos:** Product Backlog (lista de requisitos del producto), Sprint Backlog (lista de requisitos del sprint) y el Incremento.
- **Eventos:** Sprint Planning (plan de la iteración), Dailys (reuniones diarias), Sprint Review (revisión de la iteración) y Sprint Restrospective (retrospectiva de la iteración).

Kanban: El equipo Kanban deriva de los sistemas de producción Lean utilizados en Toyota. “Kanban” es una palabra japonesa que significa tablón de anuncios. En Kanban se definen etapas por la que cada división del proyecto entregable o tarea debe avanzar hasta ser completada y pone límites en el número de tareas en cada fase (WIP) para hacer más visibles los problemas y reducir los costes y las pérdidas asociados con cambios durante el desarrollo.

Programación extrema (por sus siglas XP, Extreme Programming): Mientras que el método Scrum, en el nivel de dirección del proyecto, se centra en priorizar el trabajo y obtener feedback, XP se centra en las buenas prácticas del desarrollo de software. Los valores principales de esta metodología son la simplicidad, comunicación, feedback, valor y respeto, y éstos se manifiestan en las prácticas llevadas a cabo en el ciclo de vida del proyecto XP:

ENTREGAS PEQUEÑAS - PRUEBAS DE CLIENTE - PROPIEDAD COLECTIVA DEL CÓDIGO - ESTÁNDARES DE CÓDIGO - RITMO SOSTENIBLE - INTEGRACIÓN CONTÍNUA - DESARROLLO GUIADO POR LOS TESTS - REFACTORIZACIÓN - DISEÑO SENCILLO - PROGRAMACIÓN EN PAREJAS.



Ventajas

- Adaptabilidad y flexibilidad para cambios.
- Mejora continua y retroalimentación rápida.
- Entrega frecuente de incrementos funcionales.



Desventajas

- Requiere una gestión y colaboración intensivas.
- Su implementación en equipos grandes o distribuidos puede resultar compleja.
- Necesita una disciplina estricta para mantener la calidad.

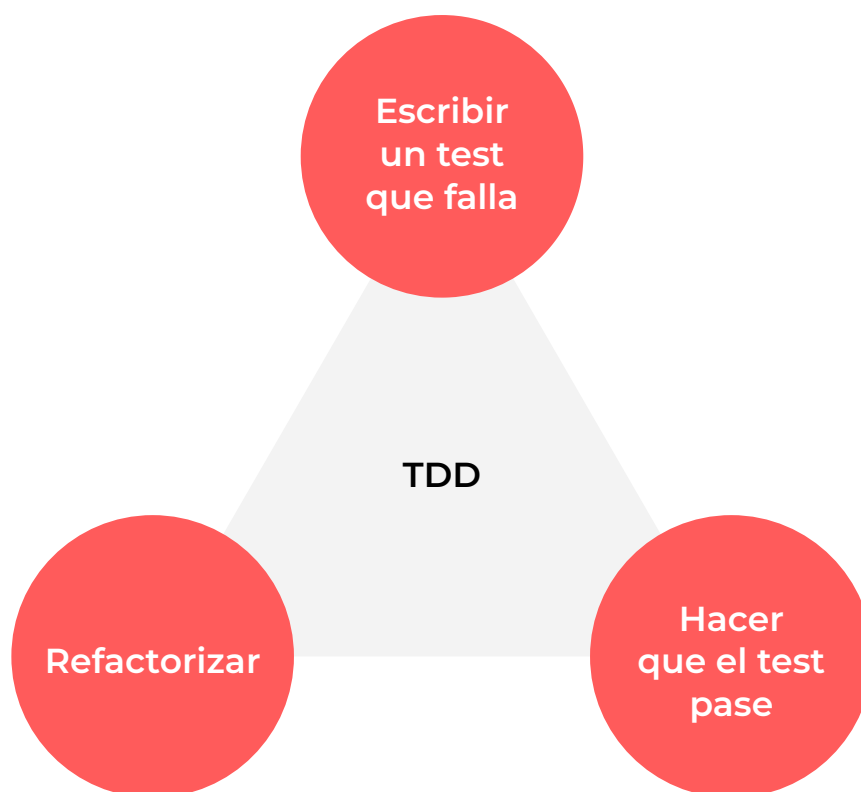
Desarrollo orientado a pruebas (TDD)

El desarrollo orientado a pruebas (TDD, por sus siglas en inglés: Test-Driven Development), es una metodología de desarrollo de software en la cual las pruebas unitarias se escriben antes de la creación del código. Este enfoque asegura que cada componente del software sea probado de manera exhaustiva desde el principio y que cualquier modificación en el código sea validada por estas pruebas.



Etapas del ciclo TDD

El proceso del TDD consiste en iteraciones sobre un ciclo de tres etapas: Red-Green-Refactor (rojo-verde-refactorizar). En cada ciclo se trabaja en una nueva hipótesis sobre el comportamiento esperado del sistema.



Veamos en qué consiste cada etapa del ciclo TDD:

Red: Se escribe una prueba automatizada para una nueva funcionalidad que falla inicialmente, ya que la funcionalidad aún no se ha implementado. El paso garantiza que la prueba es válida, falla porque el comportamiento deseado no está implementado, y necesita nuevas funciones para pasar.

Green: Se escribe el código mínimo necesario para que la prueba pase. En esta etapa del ciclo se trata de escribir el código necesario para que la prueba pase cuanto antes, sin detenerse en detalles de calidad de este.

Refactor: Una vez superado la prueba, el código se ajusta para mejorar su estructura y calidad, mientras las pruebas siguen funcionando. Esto incluye eliminar redundancias y mejorar la legibilidad.



Beneficios del TDD

TDD es una metodología poderosa para crear software de alta calidad de una manera iterativa y sistemática. Aplicar TDD en tus proyectos asegura una serie de beneficios entre los que se encuentran:

- **Mejora la calidad del código:** Al escribir pruebas antes de codificar, se asegura que el código cumple con los requisitos especificados desde el principio.
- **Fomenta un desarrollo ágil:** Aunque inicialmente el TDD puede parecer que ralentiza el proceso de desarrollo, a largo plazo lo hace más eficiente al reducir el tiempo dedicado a depurar y corregir errores.
- **Facilita el mantenimiento:** El código es más modular y mantenible gracias a las pruebas automatizadas, que funcionan como red de seguridad.
- **Detección temprana de errores:** Los errores se detectan más rápidamente porque las pruebas se ejecutan frecuentemente durante el desarrollo.
- **Fomenta el diseño de código limpio:** La necesidad de que el código pase las pruebas incentiva a los desarrolladores a escribir código más limpio y mejor estructurado.



Desafíos del TDD

TDD plantea varios desafíos que las organizaciones y los equipos de desarrollo deben superar para aprovechar plenamente su potencial. Entre estos desafíos se incluyen:

- **Curva de aprendizaje:** Los desarrolladores pueden necesitar tiempo para adaptarse a escribir pruebas antes del código.
- **Disciplina y tiempo:** Requiere una disciplina constante para escribir y mantener las pruebas, lo cual puede ser visto como un aumento en el tiempo de desarrollo inicial.
- **Cobertura de pruebas:** No todas las funcionalidades o casos de uso pueden ser cubiertos fácilmente por pruebas unitarias.
- **Mantenimiento de las pruebas:** A medida que el proyecto crece, también lo hace el conjunto de pruebas. Mantener las pruebas actualizadas con cambios en el código y nuevos requisitos puede convertirse en una tarea desafiante si no se cuenta con los roles necesarios.
- **Cambio cultural:** TDD no es solo una técnica de desarrollo; es una filosofía que requiere un cambio cultural dentro del equipo de desarrollo. Adoptar TDD significa cambiar la mentalidad de “escribir código primero” a “escribir test primero”.



Mejores prácticas de TDD

Una vez considerados sus beneficios y retos si queremos empezar a aplicarlo, surge la pregunta: ¿por dónde empezar? A continuación, explicamos algunas de las prácticas de TDD que más pueden ayudarle.

- **Pruebas pequeñas y focalizadas:** Cada prueba debe centrarse en un único aspecto o funcionalidad. Esto hace que las pruebas sean más fáciles de escribir, entender y mantener.
- **Pruebas claras y concisas:** Las pruebas deben ser claras y legibles, actuando como documentación viva del sistema. Además, tienen que dejar claro el objetivo y el uso de las unidades de código para facilitar la comprensión y el mantenimiento a largo plazo.
- **Denominación de las pruebas:** Los nombres de las pruebas deben ser descriptivos y reflejar lo que están probando, para facilitar al resto del equipo el entendimiento del propósito de la prueba y el aspecto del código que se está revisando.
- **Refactorización constante:** Refactoriza el código regularmente para mejorar su diseño y eliminar duplicaciones, con esto se simplifica el código sin comprometer la funcionalidad.
- **Automatización de pruebas:** Las pruebas deben ejecutarse tan a menudo como sea posible, idealmente de forma automática mediante la integración continua, para proporcionar retroalimentación rápida sobre los cambios en el código.
- **Pruebas independientes:** No deben depender unas de otras. Esto asegura que pueden ejecutarse en cualquier orden y que el fallo de una prueba no afecta a los demás. Por ello, es conveniente ejecutar todas las pruebas cuando se añade un comportamiento nuevo (en la fase Green) porque pasar la última prueba no significa que no exista un error en otra parte. Así nos aseguramos de que no se ha introducido ningún error de regresión.
- **Sencillez:** Las pruebas deben ser simples y evitar contener lógica condicional o bucles, ya que esto puede introducir errores en las propias pruebas y hacerlos más difíciles de comprender y mantener.
- **Disciplina:** Para que el proceso se mantenga efectivo, el equipo debe ser consecuente y seguir siempre los tres pasos del ciclo TDD (Red, Green, Refactor), sin caer en la tentación de escribir código sin una prueba fallida.
- **Programación en parejas (Pair programming):** Implementar TDD en parejas te ayuda a obtener feedback inmediato sobre las pruebas y su comportamiento. Al aplicar pair programming mejorarás tus capacidades gracias al intercambio positivo y constructivo de ideas.



Desarrollo orientado al comportamiento (BDD)

En la Ingeniería de Software, behavior-driven development o desarrollo guiado por el comportamiento, por sus siglas en inglés BDD es un proceso de desarrollo de software que surgió a partir del desarrollo guiado por pruebas. BDD es una técnica de desarrollo ágil de software que podríamos entender como una extensión de TDD, que abarca más fases del ciclo de desarrollo e involucra a más roles porque que, en BDD, los casos de prueba se escriben en un lenguaje natural que incluso los no programadores pueden leer.

BDD implica la creación de un caso de pruebas basado en una especificación o requerimiento, que debe estar centrada en el usuario y el sistema. Después se hace la implementación en código de la forma más rápida posible para hacer que esos casos de prueba pasen y por último se realiza la refactorización para eliminar duplicados. Este ciclo se repite hasta que el sistema funcione como se espera.

Fue creado por el ingeniero Dan North en el año 2006, se ha comentado en varias ocasiones que debería llamarse Desarrollo guiado por Funcionalidad, porque se orienta a las funcionalidades o características deseadas en el sistema. Lo diseñó basándose en TDD para corregir algunas de sus deficiencias, principalmente que usando solo TDD un software puede pasar correctamente todas las pruebas escritas y, aun así, no conseguir la funcionalidad deseada. En TDD, un sistema que pase todas las pruebas no cumple necesariamente las expectativas del cliente. Antes de instaurar BDD en un equipo o proyecto, se deben tomar algunas medidas, que cambiarán el foco de todo el desarrollo, ahora lo más importante es el punto de vista del usuario:

- Aprender un lenguaje común, DSL (Somain System Lenguaje o Lenguaje del dominio del sistema). Todos los participantes en el proyecto deben conocer su nomenclatura y sus reglas.
- Uso de historias de usuario: los requerimientos deben ser escritos en forma de historias de usuario, para que sea más fácil extraer los casos de pruebas desde el inicio.

Una historia de usuario, comúnmente se formula de esta manera:

Como [perfil], quiero [una característica = acción, objetivo del software], para lograr [resultado].

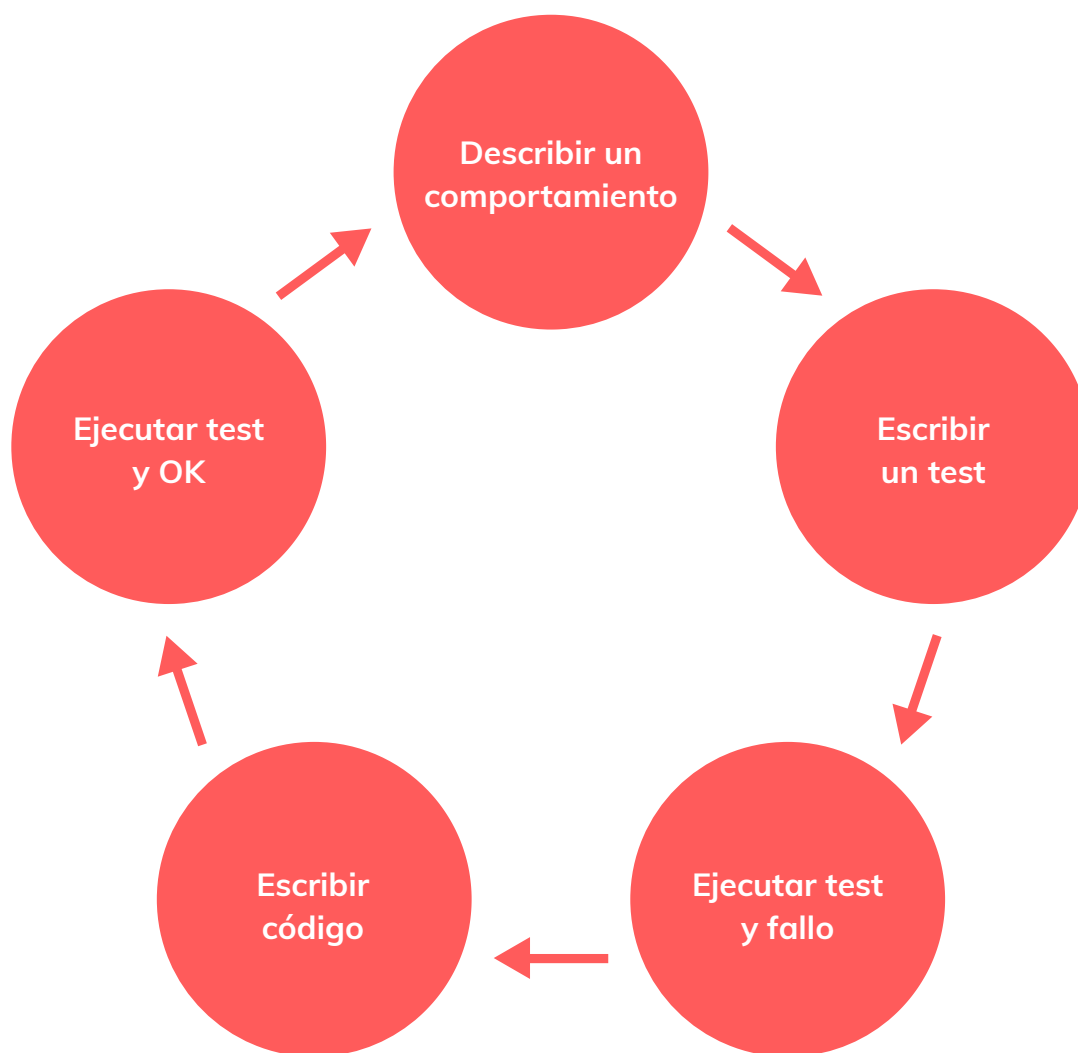
Historia de Usuario:

Como comprador, **quiero** poder añadir artículos a mi cesta de la compra, **para** poder comprar.

División de la historia de usuario en características y escenarios. Partiendo de una historia de usuario se extraen distintos comportamientos del sistema según las acciones o los datos de entrada:

Escenario inicial	Acción	Resultado esperado
Cesta vacía	Añadir el producto "El Quijote"	La cesta debe contener un artículo y el total debe ser de 45 euros.
Cesta con un producto ("El Quijote") que cuesta 45 euros.	Eliminar el artículo de la cesta.	Cesta vacía. Total de 0.
Cesta con 3 unidades del mismo producto.	Añadir otra unidad del mismo producto.	Cesta todavía con tres artículos. Se muestra un mensaje diciendo que los clientes no pueden comprar más de tres artículos del mismo producto.

El procedimiento de BDD nos recuerda mucho al de TDD, pero estamos a un nivel más alto, de diseño de la funcionalidad de aplicación y programación a más alto nivel.



Describir un comportamiento: Como hemos visto anteriormente, tras analizar la historia de usuario o requisito, el equipo trabaja en conjunto para describir comportamientos deseados en el sistema. La especificación es más clara en forma de tablas como vimos en el ejemplo anterior: se trata de conseguir un consenso en cuanto a cómo debe responder el sistema en cada situación. En esta descripción deben participar representantes de las áreas de negocio, desarrollo y calidad.

Escribir un test: en esta fase, se escriben los casos de prueba orientados a negocio, a diferencia de TDD no son tests unitarios o de clases, son pruebas funcionales a nivel de usuario, orientadas a comprobar el correcto funcionamiento de uno de los comportamientos definidos en la fase anterior. Preferentemente, estos tests deben ser automatizados, aunque no es indispensable.

Ejecutar test y fallo: se realizan las pruebas anteriormente definidas sobre el sistema existente, antes de realizar ningún cambio o adaptación. Lógicamente, estas pruebas darán un resultado de fallo.

Escribir código: ahora que el objetivo está claro, se escribe (o modifica) el mínimo código necesario para que se cumplan las condiciones de satisfacción y el test pase. En esta fase se puede utilizar TDD, es decir, hacer pequeñas iteraciones de red-green-refactor para las clases o módulos hasta tener desarrollada la funcionalidad completa.

Ejecutar test y OK: tras el desarrollo, si realizamos el mismo test, debe completarse correctamente.

Al finalizar el ciclo, vuelve a comenzar, se va iterando hasta tener el sistema funcionando completamente.



Ventajas de BDD

- Se centra en los objetivos empresariales y las necesidades del cliente.
- Establece pautas para las reuniones de revisión del diseño y análisis en equipo: las hace más eficaces y productivas.
- Facilita poder definir los criterios de aceptación antes de comenzar el desarrollo.
- Evitar el esfuerzo en características que no contribuyen a los objetivos de negocio.
- Evitar interpretaciones erróneas que lleven a rehacer el trabajo más adelante.

- Lenguaje omnipresente. Un lenguaje específico del dominio que se utiliza en todas partes en el software.
- Las especificaciones pueden ser entendidas sin problemas por todas las personas que participan en el proyecto, aunque no sean expertas en programación.
- Las especificaciones de las pruebas pueden ser ejecutadas tal cual.
- Pruebas de aceptación automatizadas que proporcionan una retroalimentación rápida.
- Especificaciones vivas que nunca se desactualizan.
- Documentación técnica y de usuario final generada automáticamente.



Diferencias entre TDD Y BDD

	TDD	BDD
 Definición	Técnica de desarrollo centrada en las características.	Controlar la calidad del software.
 Participantes	Desarrolladores.	Desarrolladores. Clientes. QAs.
 Lenguaje usado	Similar al utilizado para el desarrollo de la característica o el mismo (java, Python...).	Gherkin (inglés simple).
 Focalizado en	Test Unitarios.	Entender los requisitos.
 Herramientas	Xunit, Junit, Nunit...	Cucumber, Jbehave, Spec Flow, BeanSpeac, Concordian...

TDD es una metodología en la que las pruebas guían el desarrollo, de forma que primero se escribe la prueba unitaria que debe fallar y después se escribe el código fuente para que la prueba sea exitosa.

Mediante el uso de TDD, los desarrolladores pretenden crear un código limpio y sencillo, compuesto por módulos poco acoplados, lo que conduce a una mayor capacidad de mante-

nimiento. Por ello, TDD es una forma de trabajar que involucra directamente a los desarrolladores, pues son estos profesionales los encargados de realizar las pruebas unitarias. Como se ha visto, las pruebas unitarias pueden llevar al problema de implementar correctamente las características equivocadas.

BDD no está centrado en el desarrollador, sino que fomenta la colaboración entre los desarrolladores y otros miembros del equipo, incluso los que no tienen conocimientos de codificación. Se puede y se debe utilizar TDD y BDD juntos. Al inicio de cada iteración, se pueden crear las pruebas automáticas de aceptación basándose en la información que proporciona el experto en el dominio.

Durante el proceso de desarrollo del código que implementa las nuevas funcionalidades, se utiliza TDD para escribir el código, definiendo tests automáticos para cada clase antes de implementarla. TDD encaja aquí perfectamente dentro de la etapa de desarrollo de BDD.

BDD se concentra en la creación de un entendimiento común de la funcionalidad a desarrollar mientras que el TDD se centra en la cobertura de pruebas unitarias de todo el código.

Modelo iterativo

El modelo iterativo es un enfoque de desarrollo en ciclos repetidos, donde se construyen y mejoran versiones del producto en cada iteración. Cada ciclo incluye planificación, diseño, desarrollo, pruebas y revisión, a diferencia de las metodologías ágiles, cada iteración o ciclo está estructurado en esas fases consecutivas como si fuera un pequeño proyecto en cascada.

Este modelo es útil para proyectos donde los requisitos pueden evolucionar con el tiempo y es necesario adaptar el software.



Etapas en cada ciclo del modelo iterativo

- **Planificación:** Identificación de requisitos y objetivos para la iteración.
- **Diseño:** Creación de un diseño preliminar para la funcionalidad a desarrollar.
- **Desarrollo:** Codificación y construcción de la funcionalidad.
- **Pruebas:** Ejecución de pruebas exhaustivas para asegurar la calidad.
- **Revisión y Retroalimentación:** Evaluación de los resultados y planificación de mejoras para la siguiente iteración.



Ventajas

- Flexibilidad para incorporar cambios en los requisitos.
- Identificación y corrección temprana de errores.
- Mejora gradual y continua del producto.



Desventajas

- Gestión más compleja debido a múltiples iteraciones.
- Necesidad de una colaboración activa y constante.
- Posible sobrecarga de pruebas continuas.

Enfoque DevOps

DevOps es una combinación de “Development” (Desarrollo) y “Operations” (Operaciones) que busca eliminar las barreras entre los equipos de desarrollo de software y los equipos de operaciones. DevOps fomenta una cultura de colaboración y comunicación continua, con un fuerte enfoque en la automatización y la entrega continua.



Principios del enfoque DevOps

- **Colaboración y Comunicación:** Fomento de una cultura de trabajo conjunto entre desarrolladores, testers y operadores.
- **Automatización:** Automatización de compilaciones, pruebas, despliegues y monitoreo.
- **Integración Continua (o CI del inglés Continuous Integration):** Integración frecuente del código en un repositorio compartido con pruebas automáticas.
- **Entrega o Despliegues Continua (CD por sus siglas en inglés, Continuous Delivery):** Automatización del proceso de despliegue para lanzamientos rápidos y seguros.
- **Monitoreo Continuo:** Supervisión constante del software en producción para detectar y resolver problemas rápidamente.



Prueba continua

La prueba continua es un componente esencial de DevOps, asegurando que las pruebas se ejecuten de manera automática y continua en todo el ciclo de vida del desarrollo. Esto garantiza que los errores se detecten y se corrijan lo antes posible.



Ventajas

- Entrega rápida y frecuente de nuevas funcionalidades.
- Mejora continua de la calidad del software.
- Reducción de riesgos mediante detección temprana de errores.
- Mejor colaboración entre equipos de desarrollo y operaciones.



Desafíos

- Complejidad de implementación y mantenimiento de la infraestructura de DevOps.
- Necesidad de un cambio cultural para adoptar prácticas DevOps.
- Requiere herramientas especializadas y competencias en automatización.

2.3 Niveles de prueba

Las pruebas de software son fundamentales para el desarrollo de cualquier producto de digital, permite asegurar que el software cumpla con los requisitos y estándares de calidad.

De acuerdo con ISTQB se definen 4 niveles de pruebas que abordan diferentes aspectos del software, estos son:

Pruebas de Componentes

Pruebas de Integración

Pruebas de Sistema

Prueba de Aceptación

1. Pruebas de Componentes

En este nivel, las pruebas se enfocan en evaluar los componentes individuales del software, como funciones, módulos y objetos. El propósito es asegurar que cada componente cumpla con los requisitos y funcione según lo previsto, estas pruebas se pueden realizar de forma manual o automatizada.

2. Pruebas de Integración

Son pruebas orientadas a evaluar la interacción entre los diferentes componentes del software. El propósito es asegurar que los componentes individuales funcionen juntos correctamente. Se pueden realizar pruebas manuales o automatizadas de cada componente o funcionalidad en un contexto integrado del sistema.

3. Pruebas del Sistema

Se centran en evaluar el sistema completo, incluyendo la interacción entre los componentes y la funcionalidad general del software. El propósito es asegurar que el software cumpla con los requisitos funcionales y no funcionales. Se pueden realizar pruebas manuales o automatizadas para evaluar el rendimiento, la seguridad y usabilidad del sistema.

4. Pruebas de Aceptación

En este nivel, se evalúan aspectos del software con relación a los requisitos del negocio y el usuario final. Su propósito es asegurar que el software cumpla con los requisitos del negocio y expectativas del cliente. Se pueden realizar pruebas manuales o automatizadas para evaluar los aspectos funcionales del software en el contexto del negocio.

2.4 Tipos de pruebas

Pruebas funcionales

Las pruebas funcionales se centran en verificar que el software cumpla con los requisitos funcionales especificados y proporcionan una experiencia de usuario óptima, con su implementación identificamos defectos en etapas tempranas del desarrollo (antes de que el software llegue al usuario final), por lo que contribuye a aumentar la calidad del desarrollo.

Desde esta perspectiva, se pueden derivar varios tipos de pruebas funcionales, entre las más comunes podemos mencionar:

- **Pruebas de Interfaz de Usuario (UI):** Verifican que las interfaces de usuario funcionen correctamente según lo previsto. Evalúan la facilidad de uso y la funcionalidad de las interfaces gráficas.
- **Pruebas de integración:** Aseguran que los diferentes módulos o servicios del software interactúen correctamente entre sí. Se pueden realizar tanto a nivel de módulos individuales como a nivel de sistema completo.
- **Pruebas de regresión:** Se realizan para asegurarse de que las modificaciones o actualizaciones no han afectado las funcionalidades existentes. Es crucial cuando se liberan nuevas versiones del software.
- **Pruebas de sistema:** Evalúan el sistema completo para verificar que cumple con los requisitos especificados. Incluyen la validación del flujo de trabajo completo del software, de principio a fin.
- **Pruebas de aceptación:** Aseguran que el sistema cumpla con las expectativas del cliente o del usuario final. Pueden ser pruebas de aceptación del usuario (UAT) o pruebas de aceptación operativa (OAT).
- **Pruebas de exploración:** Los testers exploran el sistema sin casos de prueba predefinidos, en busca de fallos o defectos inesperados. Útiles para descubrir problemas en áreas que no están completamente especificadas.
- **Pruebas de localización y globalización:** Aseguran que el software sea funcional en diferentes idiomas, zonas horarias, y configuraciones regionales. Incluyen pruebas de adaptación a los diferentes entornos culturales y normativas legales.
- **Pruebas de accesibilidad:** Verifican que el software sea accesible para usuarios con discapacidades, cumpliendo con las normativas y directrices, como las WCAG (Web Content Accessibility Guidelines).
- **Pruebas de compatibilidad:** Se enfocan en verificar que el software funcione correctamente en diferentes dispositivos, navegadores, sistemas operativos, y configuraciones de hardware.

Pruebas no funcionales

Son aquellas que se encargan de verificar aspectos no funcionales ni relacionados con la lógica de negocio. Se enfocan en validar la forma en que el sistema funciona, y nos permiten conocer que riesgos corre el mismo en cuanto a desempeño y rendimiento en entornos productivos, ya que se centran en aspectos tales como:

Rendimiento

Escalabilidad

Seguridad

Usabilidad

Confiability

Estas pruebas nos permiten entender cómo se comportará el sistema una vez puesto en producción, conocer cuántos usuarios simultáneos o concurrentes soporta el sistema, cuántas solicitudes es capaz de atender, los recursos mínimos necesarios para una correcta operación, la reacción ante fallos de hardware, comportamiento ante el procesamiento de grandes volúmenes de datos, cómo reaccionará a la sobrecarga o entradas erróneas de datos no esperados, etc.

Entre los tipos de pruebas no funcionales más comunes, podemos mencionar:

- **Pruebas de carga:** Evalúa el comportamiento del sistema tanto en condiciones normales como con cargas pesadas. Sirve para determinar cuanta carga puede soportar el sistema sin verse afectado negativamente.
- **Pruebas de rendimiento:** Comprueban la velocidad de respuesta y estabilidad del sistema en su conjunto y por separado. Suelen comprobar los tiempos de respuesta, posibles cuellos de botella y puntos de fallo, entre otras cosas. Ayudan a garantizar la estabilidad, fiabilidad, velocidad y calidad del sistema.
- **Pruebas de estrés:** Son pruebas de carga que se realizan con demandas mayores a la capacidad del sistema, para intentar determinar la estabilidad de este, la disponibilidad, el manejo de errores, etc.
- **Pruebas de seguridad:** Se utilizan para verificar si un sistema seguro o no, si puede ser vulnerado, si se controla el acceso mediante cuentas de usuario, si se pueden vulnerar estos accesos, entre otras cosas.
- **Pruebas de escalabilidad:** Comprueban hasta qué punto el sistema puede ampliar o “escalar” su capacidad de procesamiento para satisfacer una demanda creciente.
- **Pruebas de recuperación:** Verifican que tan rápido y bien se recupera el sistema luego de un fallo.
- **Pruebas de volumen:** Verifican el correcto funcionamiento del sistema ante ciertos volúmenes de datos.

3 Técnicas de diseño de casos de pruebas

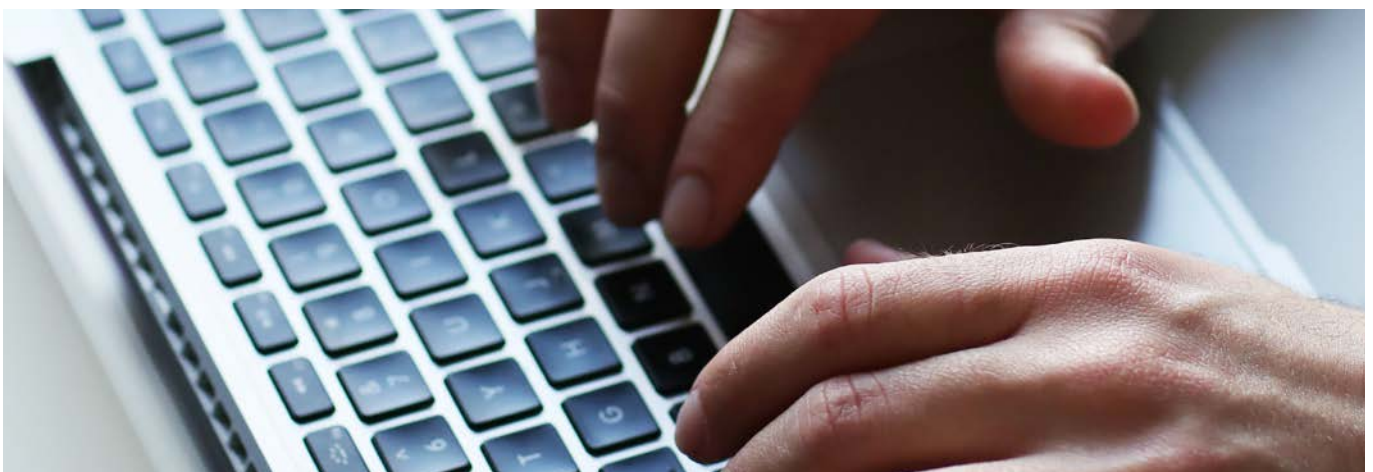
3.1 Técnica de diseño de pruebas basada en la experiencia

Introducción a las pruebas basadas en la experiencia

Las técnicas de prueba basadas en la experiencia son un conjunto de técnicas que se basan en el conocimiento, la intuición y la experiencia previa del tester para identificar problemas potenciales en el software. Estas técnicas son menos estructuradas que las pruebas formales, y a menudo se utilizan en conjunción con otros enfoques de prueba.

Se caracterizan por ser pruebas con bastante flexibilidad de ejecución, ya que permiten un enfoque dinámico y adaptable a medida que el tester aprende más sobre el sistema bajo prueba. Además, son pruebas rápidas de implementar, debido a que no requieren una extensa preparación o documentación previa, puesto que se base en el conocimiento previo y la experiencia del tester con el sistema o sistemas similares.

La problemática que pueden presentar este tipo de pruebas, son que los resultados pueden variar significativamente entre diferentes testers, al tener cierta subjetividad, además es difícil asegurar que todas las áreas del sistema se hayan probado adecuadamente. Por la naturaleza ad hoc de estas pruebas puede resultar en una documentación menos completa y replicable que en otro tipo de pruebas. Lo que hace que se requieran testers con bastante experiencia y un conocimiento profundo para que estas se consideren efectivas.



Tipos de pruebas basadas en la experiencia

1. Predicción de errores

La predicción de errores implica que el tester utilice su experiencia para predecir y probar los defectos más probables en el sistema. Esta técnica puede ser muy efectiva para encontrar defectos comunes o recurrentes, basándose en el conocimiento del tester sobre errores típicos en sistemas similares. Sin embargo, la efectividad de esta técnica depende en gran medida de la habilidad y experiencia del tester, lo que puede limitar su aplicabilidad en algunos contextos.

2. Pruebas exploratorias de forma simultánea

En las pruebas exploratorias, el tester explora el sistema de manera libre, diseñando y ejecutando pruebas en tiempo real basándose en su intuición y experiencia. Este enfoque permite descubrir defectos imprevistos y ajustar el enfoque en función de los hallazgos durante la exploración. Sin embargo, puede ser difícil asegurar una cobertura completa del sistema debido a la naturaleza ad hoc de estas pruebas.

3. Pruebas ad-hoc

Las pruebas ad-hoc son una técnica de pruebas sin especificación previa y sin planificación, se pueden ejecutar en cualquier momento del ciclo de vida del software, suelen orientarse a áreas críticas y sensibles del sistema que se está probando. Estas pruebas pueden ser útiles en situaciones en las que se requiere un rápido descubrimiento de errores, cuando se necesita obtener una retroalimentación inmediata del software o cuando se quiere explorar un nuevo conjunto de características que aún no han sido incluidas en el plan de pruebas formal.

4. Pruebas de ataque

Las pruebas de ataque consisten en que el tester intente quebrantar la seguridad sistema de manera intencionada, utilizando métodos similares a los que un atacante malintencionado podría usar. Este enfoque es útil para identificar vulnerabilidades y defectos de seguridad, asegurando que el sistema es robusto ante posibles ataques. Sin embargo, requiere un conocimiento profundo de técnicas de hacking y seguridad, lo que puede limitar su aplicación a testers con experiencia específica en este ámbito.



Estrategias para implementar pruebas basadas en la experiencia

1

Integración con otras técnicas de prueba

Trata de complementar las pruebas basadas en la experiencia con técnicas más estructuradas, para asegurar una cobertura completa.



Beneficios:

Combinar enfoques permite una cobertura más exhaustiva del sistema, aprovechando la flexibilidad y adaptabilidad de las pruebas exploratorias junto con la precisión de las pruebas estructuradas.

2

Documentación y registro

Aunque las pruebas basadas en la experiencia son menos formales, es crucial documentar los hallazgos y las rutas de prueba exploradas para asegurar la reproducibilidad y facilitar la comunicación de los hallazgos.

Para ello, se utilizan herramientas de gestión de pruebas como TestRail o Jira (X-Ray o Zephyr) para registrar las actividades de prueba y los hallazgos (Bugs). Durante las pruebas, capturar notas detalladas, capturas de pantalla y grabaciones de video cuando sea posible y con ello se generan informes detallados que describen los pasos seguidos, las condiciones probadas y los resultados observados.



Beneficios:

La documentación clara y precisa permite la reproducibilidad de las pruebas y facilita la comunicación de los hallazgos al equipo de desarrollo, mejorando la calidad general del proceso de pruebas.

3

Sesiones informativas y retroalimentación

Realizar sesiones informativas después de las pruebas para discutir los hallazgos, compartir conocimientos y ajustar las estrategias de prueba futuras.

Es importante programar reuniones regulares para revisar los resultados de las pruebas exploratorias donde discutir los defectos encontrados, las áreas de éxito y las posibles mejoras en el enfoque de pruebas y con ello tomar medidas basadas en la retroalimentación para ajustar y mejorar las estrategias de prueba.



Beneficios:

Fomenta la colaboración y el aprendizaje continuo dentro del equipo, mejorando la calidad general de las pruebas y asegurando que los hallazgos se integren efectivamente en el proceso de desarrollo.

4

Uso de herramientas de apoyo

Utilizar herramientas para capturar y gestionar la información durante las pruebas exploratorias.

Deben ser herramientas tales como TestRail para pruebas exploratorias, SBTM (Gestión de pruebas basadas en sesiones, por sus siglas en inglés “Session Based Test Management”) para gestionar sesiones de prueba o herramientas de captura de pantalla y grabación de video. Integrar estas herramientas en el flujo de trabajo de pruebas fomenta capturar y organizar información de manera estructurada.



Beneficios:

Mejora la eficiencia y precisión del proceso de pruebas, asegurando que los hallazgos se registren de manera estructurada y accesible.

3.2 Técnica de diseño de pruebas de caja blanca

La prueba de caja blanca es un tipo de técnica de pruebas de software en la que se examina la estructura interna, el diseño y el código del programa. A diferencia de las pruebas de caja negra, que evalúan las funcionalidades del sistema sin conocer su estructura interna, las pruebas de caja blanca examinan cómo el software procesa las entradas para generar las salidas. Diseñarlas suele requerir ciertos conocimientos de programación

Las pruebas de caja blanca pueden realizarse en distintas fases del ciclo de pruebas para verificar el funcionamiento del código y la estructura internos. Normalmente, las pruebas de caja blanca se llevan a cabo cuando los desarrolladores realizan pruebas unitarias y, a veces, durante las pruebas de integración.

Ventajas de las pruebas de caja blanca

Las principales ventajas que dan la realización de pruebas de caja blanca son las siguientes:

- Permiten una cobertura exhaustiva del código, asegurando que todas las líneas, ramas y condiciones lógicas se ejecuten y verifiquen.
- Identificación temprana de errores en el flujo lógico, en la estructura del código y en las condiciones lógicas.
- Ayudan a optimizar el código al identificar redundancias, código que nunca se ejecuta y partes ineficientes.
- Mejoran la seguridad del software al detectar vulnerabilidades, condiciones de carrera y posibles puntos de fallo internos.
- Permiten la verificación de la lógica de negocio y de las reglas de negocio implementadas en el código.
- Facilitan la revisión y el análisis del código fuente, lo que puede llevar a una mejora general en la calidad del código.
- Aseguran que los algoritmos y las estructuras de datos estén implementados correctamente y funcionen como se espera.
- Los desarrolladores y testers adquieren un conocimiento profundo del código y su funcionamiento interno.

Las pruebas de caja blanca son ideales para ser automatizadas, ya que se pueden integrar directamente en el proceso de desarrollo mediante herramientas de automatización.

Dificultades de las pruebas de caja blanca

Las pruebas de caja blanca requieren que el tester tenga un conocimiento detallado de la lógica interna del sistema y del código fuente. Esto implica que las pruebas de caja blanca sean realizadas casi siempre por los ingenieros y desarrolladores de software. Los testers de control de calidad, se suelen enfocar en otro tipo de pruebas, como pruebas de caja negra, puesto que no es habitual que tengan los conocimientos técnicos necesarios para realizar este tipo de pruebas. Pueden ser costosas de llevar a cabo debido a que estas deben actualizarse cada vez que se realizan cambios en el código, se deben identificar y probar todos los caminos de ejecución posibles dentro del código y asegurar una cobertura completa del código, incluyendo todas las ramas y caminos posibles. Esto puede llegar a ser un proceso intensivo en tiempo y recursos.

Las pruebas de caja blanca suelen usarse por los desarrolladores para verificar si el código funciona como debería, pero no pueden concluir que el código en funcionamiento ofrece los resultados esperados por los usuarios finales sin combinar las pruebas de caja blanca con las de caja negra. Incluso con herramientas de cobertura de código, interpretar los resultados y determinar la suficiencia de las pruebas puede ser complicado. La instrumentación del código para medir la cobertura y ejecutar pruebas de caja blanca puede afectar al rendimiento del sistema.

Características de las pruebas de caja blanca:

- **Conocimiento del código fuente:** se diseñan pruebas específicas para el código fuente
- **Enfoque en la lógica interna:** implica probar flujos de ejecución, condiciones y decisiones para evaluar que sean correctos y eficaces.
- **Diseño de casos de prueba específicos:** se seleccionan datos de entrada variados para evaluar todos los posibles flujos y condiciones para asegurar una cobertura exhaustiva.
- **Cobertura del código:** cada línea de código, flujo y condición debe ser probada.
- **Identificación de errores en el código:** identificar posibles errores lógicos, bucles infinitos y otros problemas específicos del código.

Tipos de pruebas de caja blanca

Aquí se resumen los principales tipos de pruebas de diseño de caja blanca:

Prueba de sentencia

La prueba de sentencias evalúa las partes del código que se ejecutan. La cobertura se calcula dividiendo el número de sentencias ejecutadas por las pruebas entre el número total de sentencias ejecutables en el objeto de prueba, y se expresa generalmente en porcentaje. Alcanzar una cobertura total de sentencias debe considerarse el mínimo requerido para todo el código sometido a prueba.

Prueba de decisión

La prueba de decisión evalúa los resultados de las decisiones en el código. Para ello, los casos de prueba siguen los flujos de control desde un punto de decisión; por ejemplo, en una sentencia "IF", existe un flujo de control para el resultado verdadero y otro para el falso. La cobertura se mide dividiendo el número de resultados probados entre el total de resultados en el objeto de prueba, generalmente expresado como porcentaje. A diferencia de otras técnicas, la prueba de decisión considera la decisión totalmente y evalúa solo los resultados verdaderos y falsos, independientemente de su estructura interna. Este nivel de cobertura es esencial cuando se prueba código que es importante o incluso crítico.

Prueba de condición/decisión modificada

La prueba de condición/decisión modificada examina cómo se estructura una decisión con múltiples condiciones. Esta técnica verifica que cada condición atómica influya de manera independiente y correcta en el resultado de la decisión global. Cada predicado de decisión se compone de una o más condiciones atómicas, las cuales se evalúan como verdaderas o falsas. Estas condiciones se combinan lógicamente para determinar el resultado de la decisión. La prueba de condición/decisión modificada asegura que cada condición atómica impacte de forma independiente y precisa el resultado global de la decisión. Esta técnica proporciona un nivel de cobertura más robusto que la cobertura de sentencias y decisiones cuando se trata de decisiones con múltiples condiciones.

Mejores prácticas para pruebas de caja blanca

Las pruebas de caja blanca implican una inspección detallada del código fuente para asegurar su correcto funcionamiento. Las mejores prácticas en este contexto se centran en la comprensión del código, la cobertura de pruebas, el uso de herramientas, y la automatización y revisión continua. Por lo que, antes de escribir las pruebas, es fundamental entender completamente el código, su estructura y los flujos de control. La documentación debe estar actualizada para facilitar este proceso.

Es esencial alcanzar una alta cobertura de código, incluyendo la cobertura de sentencias, ramas y condiciones. Esto asegura que todas las partes del código se evalúan durante las pruebas, para ello, utilizar herramientas automatizadas como JaCoCo para medir la cobertura de código. Los informes generados por estas herramientas ayudan a identificar áreas no cubiertas y a mejorar las pruebas.

Como punto también importante, es realizar revisiones de código y de pruebas regularmente para asegurar que las pruebas sean adecuadas y se mantengan actualizadas con los cambios en el código. Las revisiones por pares aportan una perspectiva adicional y mejoran la calidad de las pruebas, al igual que ejecutar pruebas de regresión regularmente para asegurar que los cambios recientes no introducen defectos en las partes ya probadas del código. Estas pruebas deben estar automatizadas en el pipeline de CI.

Para finalizar, mantener documentados los casos de prueba y generar informes detallados de los resultados para un seguimiento efectivo y para facilitar la comprensión.

3.3 Técnica de diseño de pruebas de caja negra

Una prueba de caja negra es una técnica de pruebas de software en la que el tester evalúa la funcionalidad del software sin conocer la estructura interna del código o la lógica de implementación. Esto no solo se refiere a no conocer el código fuente en sí, sino que implica no ver las documentaciones de diseño que rodean al software. Este tipo de prueba se centra en verificar que el software cumple con los requisitos y que produce las salidas correctas para un conjunto dado de entradas, por lo que los responsables de las pruebas deben ser profesionales que no hayan desarrollado el código.

El enfoque de estas pruebas es evaluar la funcionalidad del software desde la perspectiva del usuario y para ello utilizan casos de prueba que simulan escenarios de uso real.



Ventajas de las pruebas de caja negra:

- ***Perspectiva del usuario final.*** El principal objetivo de las pruebas de caja negra es conocer cuáles son los problemas de una aplicación cuando un usuario la utiliza en el día a día. El tester, para ello, evalúa el sistema desde el punto de vista del usuario, asegurándose de que la funcionalidad cumpla con sus expectativas.
- ***Sin necesidad de conocimientos técnicos.*** Las pruebas de caja negra tienen como finalidad examinar cómo funciona la aplicación para un usuario final, y el usuario estándar, por lo general, no tiene conocimientos técnicos avanzados en la mayoría de las situaciones, por ello, los testers no necesitan conocimientos técnicos profundos del código fuente, lo que permite una evaluación imparcial y una reducción en el coste de las pruebas.



Desventajas de las pruebas de caja negra:

- ***Cobertura limitada del código.*** Con las pruebas de caja negra no se pueden garantizar que todas las rutas de ejecución del código sean probadas.
- ***Automatización más compleja.*** Dado que lo que se pretende es reproducir la forma en que un usuario interactúa con un sistema, puede resultar más difícil automatizar un proceso de pruebas de caja negra. Crear y mantener conjuntos de datos de prueba robustos y representativos para cubrir todas las posibles variaciones pueden ser complejo. Además, los cambios en los requisitos de datos pueden requerir actualizaciones frecuentes de los casos de prueba.
- ***Dificultad para encontrar las causas del problema.*** Debido a que el tester, en las pruebas de caja negra, no suele tener conocimientos del código interno, uno de los principales inconvenientes es no poder detectar errores en la lógica interna del código que, de primeras, no se manifiestan en la interfaz del usuario.

Tipos de pruebas de caja negra

Partición de equivalencia

Divide las entradas en clases de equivalencia donde se espera que el sistema se comporte de manera similar, el objetivo es reducir el número total de casos de prueba al seleccionar un valor representativo de cada clase.

Análisis de valores límite

Se enfoca en probar los límites extremos de las clases de equivalencia, ya que los errores a menudo ocurren en los bordes de los rangos de entrada.

Pruebas de tabla de decisión

Los casos de uso describen interacciones entre el usuario y el sistema para lograr un objetivo específico, por lo que, las pruebas se diseñan basándose en estos casos de uso, asegurando que el sistema soporte todos los escenarios de usos descritos.

Pruebas exploratorias

El tester explora la aplicación de manera dinámica y ad hoc, diseñando y ejecutando pruebas sobre la marcha sin guiones predefinidos, basándose en la experiencia y el conocimiento para descubrir fallos que no serían identificados por casos de prueba formales.

Características de las pruebas de caja negra:

- ***Independencia del código fuente.*** Para realizar las pruebas de caja negra, el tester no necesita acceso al código fuente, esto permite una evaluación imparcial de la funcionalidad desde la perspectiva del usuario final.
- ***Basadas en requisitos y especificaciones.*** Las pruebas de caja negra se fundamentan en los requisitos y especificaciones funcionales del software. Aseguran que el software cumpla con las expectativas y necesidades del usuario y este alineado con las especificaciones requeridas.
- ***No requieren conocimientos técnicos avanzados.*** Esto permite que los testers puedan realizar pruebas de caja negra efectivas sin experiencia en programación, lo que

fomenta la colaboración entre equipos con múltiples niveles de conocimiento técnico.

- **Pruebas en etapas finales.** Las pruebas de caja negra se suelen realizar en las últimas fases del proceso de desarrollo cuando la interfaz de usuario ya esté completa y funcional. Esto se debe, a que su objetivo es verificar que todas las funcionalidades requeridas estén presentes y operativas, sin importar cómo estén implementadas internamente.
- **Uso de técnicas específicas de pruebas.** Para garantizar una cobertura adecuada de los posibles escenarios, se emplean técnicas como partición de equivalencia, análisis de valores límite, tabla de decisión, transición de estados y pruebas basadas en casos.

3.4 Especificación de casos de prueba

Introducción al diseño de casos de prueba

Un caso de prueba es un conjunto de acciones que se ejecutan para verificar una funcionalidad específica de una aplicación de software. Su diseño consiste en crear escenarios que simulan diferentes condiciones de uso, para verificar que el software funcione correctamente bajo múltiples situaciones.

Un caso de prueba bien diseñado permite evaluar aspectos como funcionalidad, rendimiento, seguridad o usabilidad del sistema, asegurando que el producto final cumpla con los requisitos y expectativas de los usuarios.

Componentes de un caso de prueba:

Identificador

Cada caso de prueba debe tener un identificador único para su referencia. Puede ser numérico o alfanumérico.

Descripción

Especifica que se probará. En algunos casos, también se incluye el entorno de pruebas, los datos y las precondiciones.

Datos de prueba

Información específica necesaria para ejecutar el caso de prueba.

Precondición

Estado en el que se debe encontrar el sistema antes de poder ejecutar los pasos siguientes.

Pasos para la ejecución

Instrucciones detalladas para la ejecución del caso de prueba.

Postcondiciones

Estado en que queda el sistema tras ejecutar los pasos.

Resultados esperados

Indica al tester los resultados obtenidos después de ejecutar los pasos y así determinar si la prueba tuvo éxito o pasó.



Enfoques de diseño de casos de prueba

1

Diseño basado en requisitos

Esta metodología se centra en crear casos de prueba a partir de los requisitos funcionales y no funcionales del software. Asegura que cada requisito tenga uno o más casos de prueba que verifiquen su implementación correcta, la cobertura de todas las funcionalidades especificadas y facilita la trazabilidad de las pruebas a los requisitos.



Beneficios:

- Asegura que todas las funcionalidades especificadas sean verificadas.
- Mejora la trazabilidad y el control de calidad.
- Facilita la identificación de requisitos no cumplidos.

2

Diseño basado en riesgos

Se centra en identificar y priorizar las áreas del sistema que tienen el mayor riesgo de fallar. Se diseñan casos de prueba específicos para cubrir estas áreas, asegurando que los problemas críticos se identifiquen y resuelvan primero.



Beneficios:

- Enfoca los recursos de prueba en las áreas de mayor riesgo.
- Aumenta la probabilidad de identificar problemas críticos temprano.
- Mejora la gestión de riesgos del proyecto.

3

Diseño basado en casos de uso

Utiliza casos de uso para desarrollar casos de prueba que simulan escenarios de uso real del software. Esto asegura que el sistema funcione correctamente desde la perspectiva del usuario final.



Beneficios:

- Mejora la cobertura desde la perspectiva del usuario.
- Asegura la funcionalidad del sistema en situaciones reales.
- Facilita la identificación de problemas de usabilidad.

4

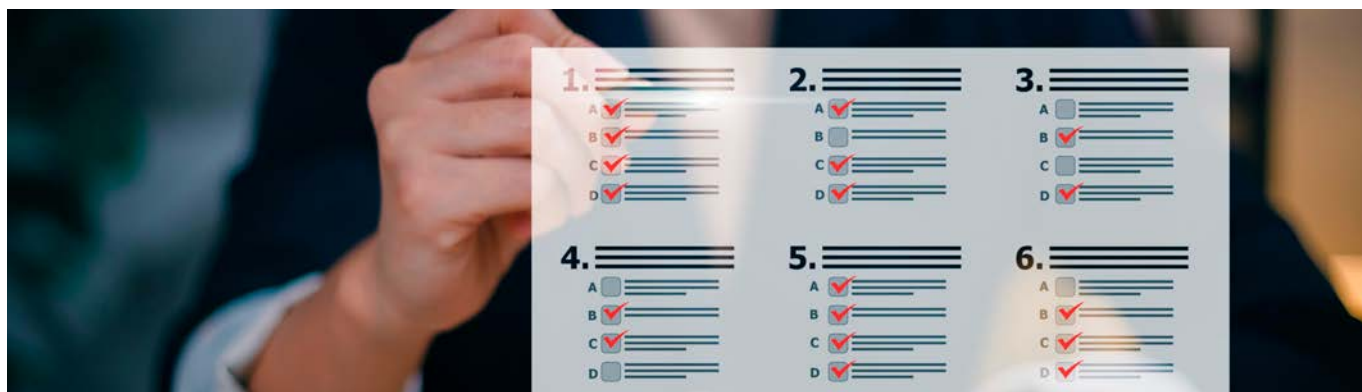
Diseño exploratorio

El diseño exploratorio implica el diseño y ejecución de casos de prueba de manera ad hoc. Los testers exploran el sistema libremente, basándose en su experiencia y conocimiento para identificar defectos.



Beneficios:

- Identifica defectos imprevistos.
- Aumenta la cobertura de pruebas en áreas no anticipadas.
- Permite una evaluación rápida y flexible del sistema.



Prácticas recomendadas en el diseño de casos de prueba funcionales

El diseño y creación de casos de pruebas funcionales son una parte integral de las actividades esenciales que desempeñan los equipos de calidad durante el ciclo de vida del desarrollo, permiten validar que el software cumple con los requisitos y funciona según lo previsto. En este apartado abordaremos algunas buenas prácticas que recomendamos adoptar desde el inicio de esta actividad para asegurar la robustez y eficiencia de la batería de pruebas, entre ellas:

Claridad y alcance de los requisitos del software

Es fundamental tener un amplio conocimiento de los requisitos del software, entender la funcionalidad y su alcance: Casos de uso, criterios de aceptación y flujos posibles, cuanto mayor sea nuestra comprensión más efectivos serán los casos de prueba, podremos validar el comportamiento previsto y detectar posibles defectos.

Crear una colección de casos de prueba completa

Es esencial que cada funcionalidad del software se pruebe de manera rigurosa. Para ello deben crearse casos de prueba que cubran diferentes escenarios, flujos de trabajo y condiciones extremas. No se debe suponer que una funcionalidad está libre de errores simplemente porque parece sencilla. La creación de casos de prueba detallados y exhaustivos ayuda a disminuir la probabilidad de que los errores lleguen a producción.

Diseñar casos de pruebas independientes y repetibles

Cada caso de prueba debe evaluar una funcionalidad o comportamiento específico sin depender en lo posible de otros casos de prueba. También debemos asegurar que estas pruebas sean repetibles, es decir, que podamos iterar en su ejecución varias veces y que podamos obtener resultados consistentes, es clave para identificar posibles defectos e identificar cuáles son susceptibles a automatización en lo sucesivo.

Incluir casos de pruebas con valores límite y datos maliciosos

Además de los casos de prueba que satisfacen los flujos básicos, se deben incluir casos con valores límite y datos maliciosos. Los casos con valores límites evalúan cómo el software gestiona los valores de entradas en los extremos del rango esperado. Los casos de pruebas con datos maliciosos permiten explorar cómo se comporta el software ante entradas incorrectas o maliciosas, para identificar posibles defectos y vulnerabilidades de seguridad.

Uso de datos de prueba realistas y representativos

Los datos de prueba son clave en el diseño de casos funcionales, conviene utilizar conjuntos de datos representativos y desde una perspectiva real, lo que permitirá validar el comportamiento previsto y detectar defectos que podrían no evidenciarse con datos ficticios, identificando problemas de escalabilidad y rendimiento.

Mantenimiento de casos de pruebas

Establecer un plan de actualización de la batería de casos de pruebas funcionales, sobre todo aquellos más relevantes para el negocio. El desarrollo de software suele ser dinámico y cambiar debido a la implementación de actualizaciones, correcciones, mejoras o simplemente la inclusión de nuevas características. Por tanto, si un caso de prueba ya no es válido por un cambio, debe modificarse por uno nuevo o quitarse de nuestra batería de pruebas.

Valorar la prioridad y aportación al núcleo del negocio

Es cierto que la automatización de casos de pruebas puede darle un plus de eficiencia a la ejecución de las pruebas durante el desarrollo en proyectos a mediano y largo plazo, no obstante, se debe tener presente qué automatizar y cuando hacerlo. Es conveniente adoptar este tipo de enfoques una vez que la funcionalidad esté estable, valorar la prioridad y aporte al "Core" del negocio, y tener en cuenta que no todas las pruebas deben ser automatizadas, las pruebas manuales aún son necesarias para evaluar otros aspectos como la experiencia del usuario entre otras áreas.

Inclusión de casos de pruebas en diferentes entornos

Es importante asegurar la existencia de casos de prueba centrados en diferentes entornos, tanto a nivel de sistemas operativos, como navegadores web y dispositivos. Para garantizar la compatibilidad y el rendimiento esperado sea consistente. Las diferencias en el entorno pueden revelar defectos que no se detectarían si probamos siempre en el mismo entorno de pruebas.

Organizar y priorizar casos de prueba

Durante la fase de diseño de casos de prueba es recomendable establecer ciertos estándares de organización de casos de prueba, lo cual permitirá categorizarlos adecuadamente y según su impacto en la funcionalidad priorizar su ejecución, automatización y gestión en general.

Técnica de ejecución de pruebas

Pruebas manuales

Estas pruebas implican que un tester valide manualmente aspectos del software, como funcionalidad o diseño de interfaz de usuario. El tester introduce los datos interactuando en forma directa con el software, y evalúa la respuesta para verificar que funcione correctamente. Estos casos de prueba suelen abarcar varios escenarios, como casos límite, de uso y condiciones de error, entre otros.

Las pruebas manuales exploratorias ofrecen una mayor flexibilidad para explorar y diseñar los casos de prueba ya que están fuertemente basadas en la experiencia de usuario y la funcionalidad del sistema. Suelen ofrecer muy buenos resultados para detectar problemas visuales en la interfaz gráfica, y resultan una herramienta eficaz para hacer pruebas iniciales o en fases exploratorias.

Por su naturaleza, estas pruebas están expuestas a errores humanos, ya que se pueden omitir en un ciclo de prueba, cometer un error al ingresar datos o evaluar las respuestas, etc.



Ventajas

- Mayor flexibilidad para explorar diferentes escenarios y casos de uso.
- Detecta problemas visuales y de usabilidad que pueden pasar desapercibidos en pruebas automatizadas.
- Ideal para pruebas iniciales y exploratorias.



Desventajas

- Requiere tiempo y recursos humanos.
- Menos confiable debido a errores humanos.
- Difícil de repetir y mantener a medida que el software evoluciona.

Buenas prácticas en pruebas manuales

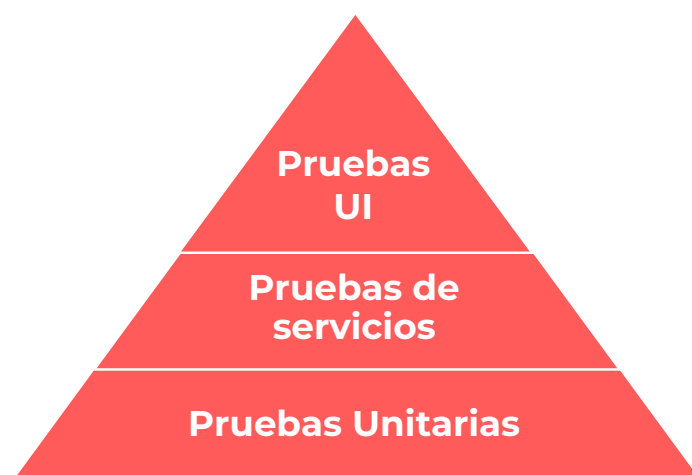
Algunas recomendaciones y buenas prácticas que se deben considerar al incluir pruebas manuales como parte de la estrategia de calidad son:

- **Enfocarse en la claridad:** Es crucial enfatizar la claridad durante todo el proceso de pruebas manuales. Ser lo más claro posible minimiza la posibilidad de malentendidos entre departamentos y profesionales, ayudando a que todos se concentren en las áreas correctas del software. Esto incluye redactar casos de prueba claros para que los pueda reproducir el tester, anotar los resultados de manera sencilla y comprensible, y asegurar que todos los miembros del proyecto comprenden el propósito y alcance.
- **Revisar todo el proceso de prueba lo más frecuente posible:** Verificar que siguen funcionando como se espera y evaluar progreso.
- **No limitarse a cazar errores:** A menudo se piensa que el principal objetivo de las pruebas de software es encontrar errores, pero esto es solo una parte del proceso, se trata de asegurar que la aplicación funcione a un alto nivel, que opere de manera predecible y sea cómoda para el usuario.

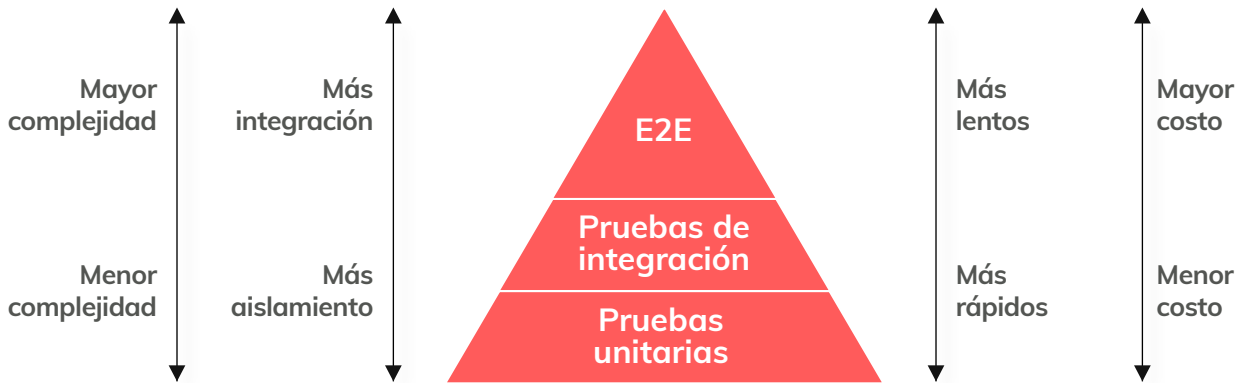
Pruebas automatizadas

Este tipo de pruebas utilizan scripts y herramientas para ejecutar pruebas automáticamente. Al realizarse de forma automatizada, se reduce el tiempo requerido para ejecutarlas. Las pruebas siguen instrucciones que imitan los pasos de un usuario, sistema o módulo realizan con la pieza de software probada, buscan y analizan inconsistencias entre las salidas obtenidas y las esperadas y, si hay diferencias, las reportan inmediatamente.

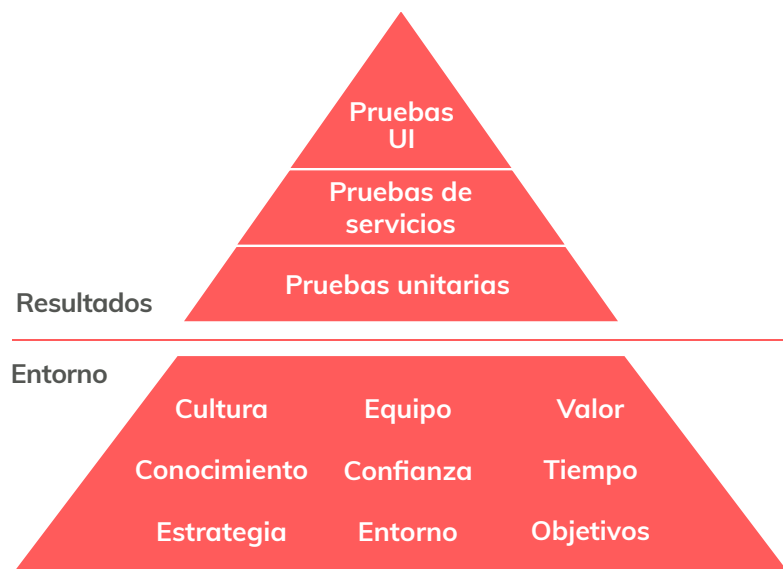
Hay distintos tipos de pruebas, que se diferencian principalmente en la complejidad, el grado de integración y el tiempo de ejecución de estas. La cantidad de pruebas de cada tipo que se deberían realizar en un sistema queda claramente representada en la “pirámide de pruebas” de Mike Cohn:



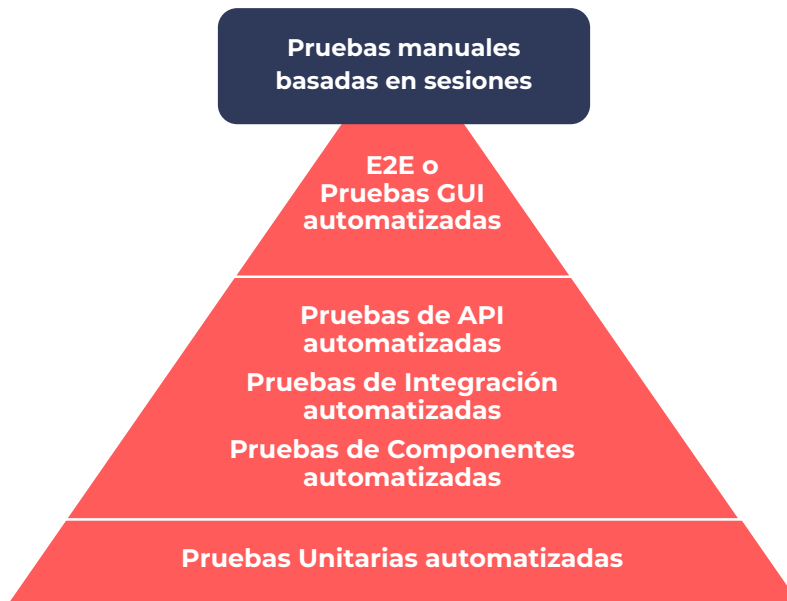
Una buena distribución de pruebas tendrá, aproximadamente, un 70% de pruebas unitarias, un 20% de servicio o integración y un 10% de UI o E2E. Como dijimos anteriormente, esto dependerá del contexto, del software a probar, de las necesidades del producto, etc., pero lo importante es entender que las proporciones deben respetar la forma de la pirámide debido a que mientras más pruebas tengamos de los niveles superiores, más complejas serán, más tiempo tomará ejecutarlas y, por lo tanto, serán más costosas.



Para que este tipo de pruebas incremente considerablemente su efectividad, tenemos que hablar de un nivel inferior adicional: para sentar las bases o los cimientos para una edificación robusta, es necesario preparar el terreno antes. Este terreno está compuesto no por habilidades técnicas o duras, si no por habilidades blandas y una cultura de empresa que haga foco en la necesidad de una buena estrategia de pruebas que nos permita asegurar un excelente nivel de calidad, a la vez que nos brinde la confianza necesaria para poder evolucionar y avanzar en el desarrollo de forma rápida y segura. Esto se logra creando una cultura entre los equipos para crear valor, compartir conocimiento y generar confianza, a la vez que se crea un entorno con objetivos y estrategias claras que perduren en el tiempo.



Al igual que las pruebas manuales, las pruebas automatizadas no son excluyentes y se pueden -y recomienda- combinar con otros tipos de pruebas, como las manuales. Esto permite crear una estrategia de QA sólida y robusta que permita obtener una alta calidad a la vez que permite un desarrollo y evolución más ágil de nuestro producto de software.



Ventajas

- Mayor velocidad y eficiencia al ejecutar casos de prueba repetitivos.
- Puede ejecutar pruebas en paralelo en diferentes entornos.
- Detecta rápidamente fallos de regresión y errores.



Desventajas

- Costo inicial de desarrollo y mantenimiento de scripts.
- Limitada para pruebas visuales y exploratorias.
- No puede reemplazar completamente las pruebas manuales.

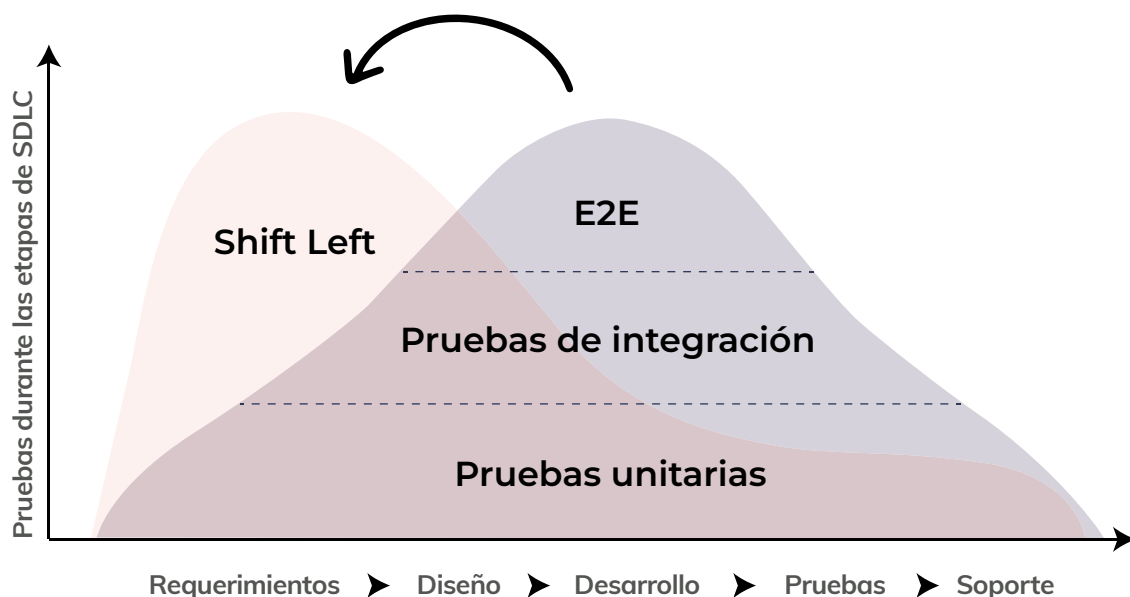
Pruebas manuales vs pruebas automatizadas

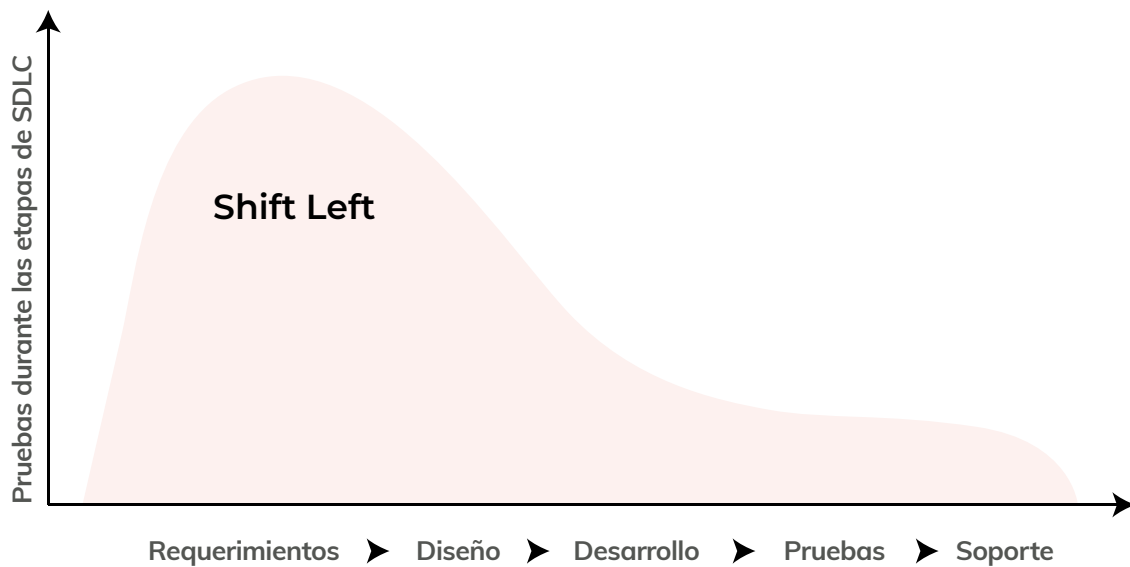
Las pruebas manuales son ideales para explorar y validar aspectos subjetivos, mientras que las pruebas automatizadas son excelentes para tareas repetitivas y regresiones. Combinar ambos enfoques proporciona una cobertura más completa y equilibrada en el proceso de pruebas. En resumen, no hay un enfoque “mejor” en general; depende del contexto y los objetivos del proyecto. La elección entre pruebas manuales y automatizadas debe basarse en las necesidades específicas del equipo y el software que se está probando, y del contexto de la empresa.

Enfoque “Shift Left” y “Shift Right” para pruebas ágiles

Enfoque Shift Left (Desplazamiento a la izquierda)

Este enfoque lo acuñó Larry Smith y tiene raíces en el desarrollo de software ágil. La idea principal detrás de este es “fallar temprano, fallar barato” y consiste en detectar los defectos y errores más temprano y más frecuentemente de forma que sean más fáciles de corregir, más barato y menos estresante para los programadores. Para ello se enfoca en realizar el grueso de las actividades de prueba más temprano en el proceso de desarrollo. La idea es “mover” las pruebas a etapas más tempranas en lugar de aguardar a las etapas finales. Este enfoque proactivo busca identificar y resolver los defectos antes, evitando que se propaguen a través de la aplicación, reduciendo el costo y el esfuerzo requerido para resolver incidencias. Por lo tanto, se le puede considerar cómo la respuesta al problema de acelerar el desarrollo de software sin comprometer la calidad. Al decir que es el resultado de realizar la mayor cantidad de pruebas al inicio del ciclo de vida del desarrollo de software (SDLC), sería el equivalente a “inclinarse” la pirámide de pruebas hacia la izquierda, de ahí su nombre.





El enfoque Shift Left incrementa la colaboración entre los desarrolladores y los testers, y ayuda también a identificar aspectos clave que se necesitan probar en etapas tempranas del desarrollo.



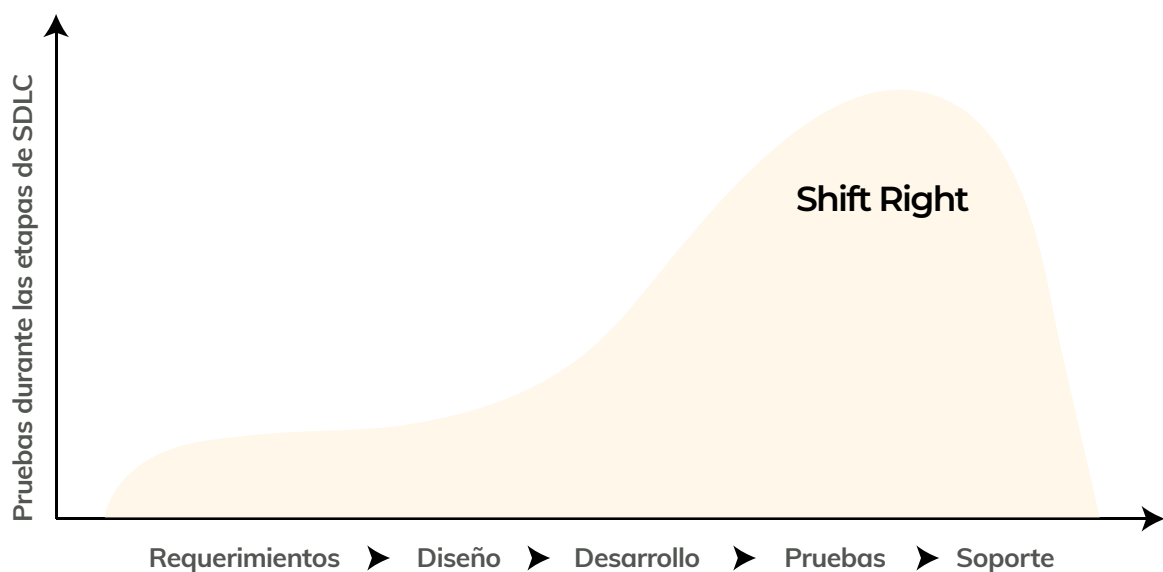
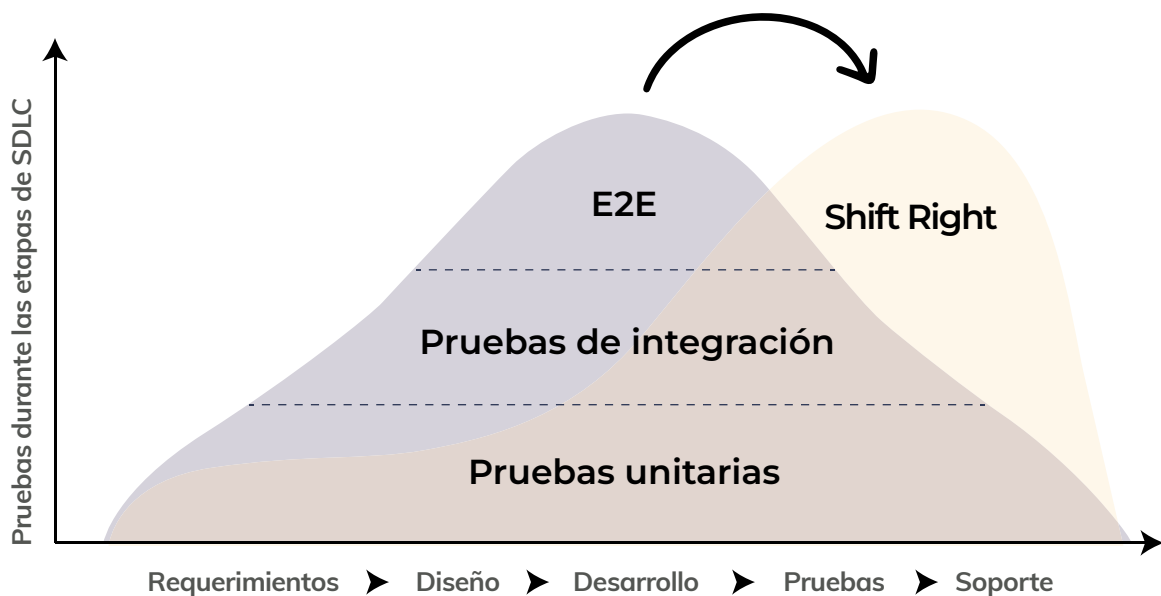
Ventajas

- **Detección temprana de errores:** al integrar el grueso de las pruebas en las etapas más tempranas del desarrollo, permite detectar los errores más temprano.
- **Reducción de costos:** al detectar los errores en las etapas más tempranas, es menos costoso resolverlo ya que no llegaron al producto final, hay menos fases o tareas que necesitan repetirse porque aún está en desarrollo.
- **Reducción de los tiempos de mercado:** descubrir un error en una etapa avanzada del proceso de desarrollo puede suponer grandes cambios en el software, con las consiguientes demoras en los plazos planificados.
- **Mejora en la cobertura de las pruebas:** al incluirse la ejecución desde el inicio del proceso de desarrollo, se pueden evaluar rápidamente todas las características, funcionalidades y performance del software.
- **Aumento del nivel de calidad del software:** las pruebas continuas e iterativas a lo largo del ciclo de vida del desarrollo de software aseguran una alta calidad del software.
- **Mejoras en la colaboración de los equipos:** al fomentar una fuerte colaboración entre equipos de desarrollo, pruebas y participantes.

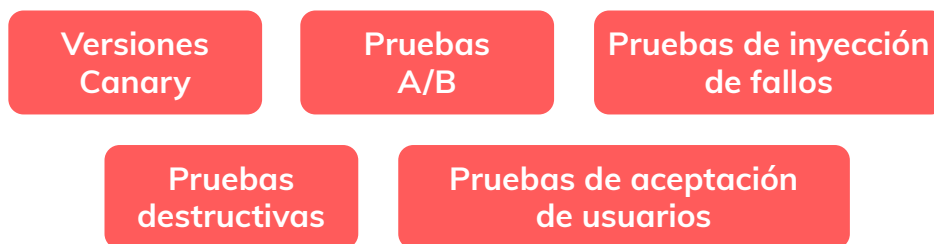
Enfoque Shift Right (Desplazamiento a la derecha)

Este enfoque se centra en realizar el grueso de las actividades de pruebas funcionales y de performances tras el proceso de desarrollo. Involucra la recopilación de información a partir de los comentarios y las interacciones reales de los usuarios una vez desplegado el software. Este enfoque permite a los desarrolladores descubrir nuevos e inesperados escenarios que podrían no haber sido detectados en el entorno de desarrollo. El objetivo de las pruebas Shift Right es asegurar que los sistemas ejecutando en producción puedan soportar una carga de usuario real, asegurando los mismos altos niveles de calidad.

Así como el enfoque Shift Left sería el equivalente a “inclinarse” la pirámide de pruebas a la izquierda, el enfoque Shift Right lo sería a “inclinarse” la pirámide a la derecha:



Algunas pruebas utilizadas en este enfoque son:



Este enfoque es importante para asegurar que una aplicación funcionará bajo cualquier circunstancia y entorno, y es crucial para garantizar la calidad del software, ya que permite recibir retroalimentación de los usuarios y el análisis de casos de uso que no se pueden prever.

Con Shift Right, los equipos pueden probar el código en entornos que imitan las condiciones reales de un ambiente productivo que no pueden replicar en entornos de desarrollo. Para automatizar parte del proceso, se pueden utilizar llamadas por API.



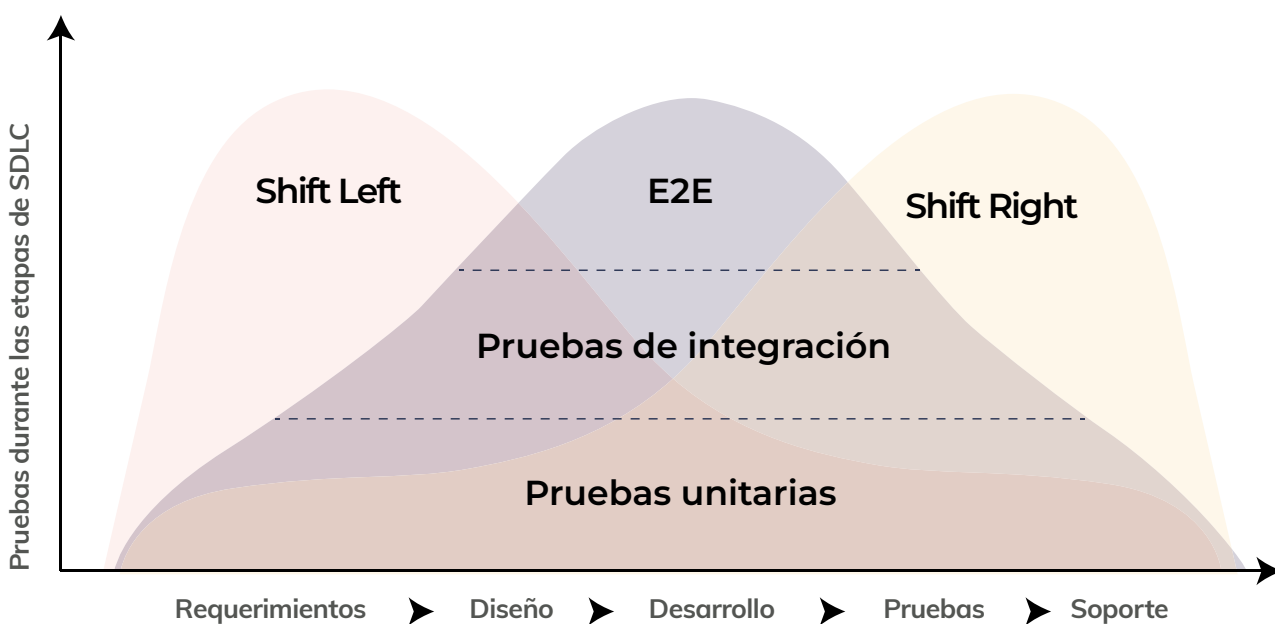
Ventajas

- **Detección temprana de problemas**, incluso antes de que los detecte el usuario, lo que permite resolverlos antes de que afecte a los usuarios reales.
- **Pruebas de usuario realistas**, que simulan que los usuarios usan el software, lo que permite detectar problemas de la experiencia de usuario y la performance que pueden haber pasado desapercibidos en pruebas anteriores.
- **Feedback continuo** entre desarrolladores y los equipos de operaciones, lo que permite ajustes rápidos, haciendo que el proceso de desarrollo sea más ágil.
- **Cobertura de pruebas**, este enfoque se complementa con métodos y enfoques tradicionales al incluir entornos de producción reales, ayudando a detectar problemas relacionados con la escalabilidad, fiabilidad y compatibilidad.
- **Monitorización mejorada**, lo que permite identificar incidencias ocasionadas por la performance, seguridad y otros problemas operativos, lo que permite construir un sistema más estable.
- **Centrado en el cliente**, ya que hace foco en lo que éste quiere, asegurando que el software cumple con los requerimientos y brinda al usuario una buena experiencia.

Conclusiones

Si bien hay algunos enfoques que en la práctica son más utilizados que otros, esto no quiere decir que sean mejores. Cada enfoque es más útil para casos concretos en los que otros enfoques podrían no encajar tan bien. La elección de enfoques dependerá del tipo de software a probar, de las necesidades de las empresas y los equipos de desarrollo y de los objetivos que se necesiten alcanzar.

Lo importante es que estos enfoques no son mutuamente excluyentes, y pueden utilizarse en conjunto para lograr una mayor cobertura y calidad, con una excelente experiencia de usuario y performance. Shift Left permite detectar la mayor cantidad de errores en etapas más tempranas del ciclo de vida de desarrollo, implementar adicionalmente el enfoque Shift Right no dispararía tanto los costos ya que la mayoría de los errores ya estarían resueltos antes de ejecutar las pruebas Shift Right, lo que permite que estas sean más concretas y específicas.



4 Gestión de actividades de prueba



4.1 Estrategia y planificación de pruebas

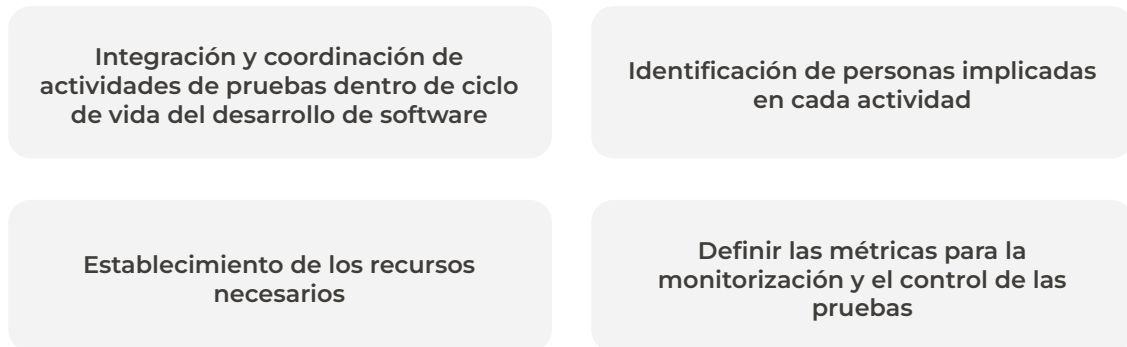
La **estrategia de pruebas** es un documento que detalla, a alto nivel, el enfoque y los objetivos para un proyecto o producto. Define el alcance, los riesgos, los recursos, las técnicas, las herramientas y los estándares que guiarán el proceso de prueba. Este documento lo suele crear el gerente o líder de pruebas en colaboración con las partes interesadas y el equipo de desarrollo, alineándose con los objetivos de comerciales y las expectativas de calidad del cliente.

El **plan de pruebas** es un documento que describe los objetivos de prueba que se deben lograr, así como los medios y el calendario para alcanzarlos, organizada para coordinar las actividades de prueba. Especifica detalladamente las actividades de prueba, las tareas, los entregables, los cronogramas y las responsabilidades para una fase o ciclo de prueba específico. Describe los casos, los datos, el entorno, las herramientas, los métodos y los criterios de prueba que se usarán para verificar la calidad del software. Este plan se deriva de la estrategia de pruebas y se actualiza a medida que avanzan las pruebas y surgen nuevos requisitos.

Al momento de determinar la planificación de las pruebas, se suele tener en cuenta:

- Matriz de responsabilidades y tareas
- Cronograma
- Condiciones para el cumplimiento del cronograma
- Potenciales riesgos y planes de respuesta en base a los requisitos establecidos. Desde factores externos que puedan tener un impacto en el proyecto, hasta problemas en la disponibilidad de entornos o de personal.

Adicionalmente, un plan de pruebas contendrá los siguientes elementos:



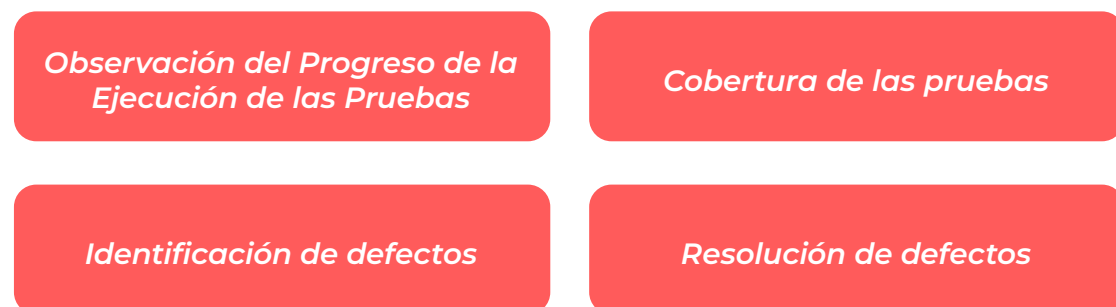
4.2 Monitorización y control de pruebas

Monitorización de pruebas

La monitorización de las pruebas es una fase de las pruebas de software que implica la supervisión y evaluación continuas de las actividades de prueba para verificar su eficacia y alineación con los objetivos del proyecto. Es un proceso de evaluación y retroalimentación de las actividades de prueba en curso. Implica comparar el estado actual de las tareas con el plan predefinido y analiza si el proceso está proporcionando los resultados esperados. Es una parte integral del ciclo de vida de las pruebas de software y de los procesos de aseguramiento de calidad.

La monitorización de las pruebas ayuda a las partes implicadas a mantenerse informadas sobre los esfuerzos de las pruebas y la calidad del software que se está desarrollando.

Esta fase implica:



Esta fase abarca el seguimiento de métricas críticas, su comparación con criterios predefinidos y la toma de decisiones informadas basadas en los datos recopilados. La supervisión de las pruebas aporta información valiosa sobre la calidad del software examinado, ayuda a detectar riesgos potenciales y permite ajustar a tiempo las estrategias de pruebas o la asignación de recursos. Garantiza que los esfuerzos de las pruebas siguen su curso, los problemas se resuelven con prontitud y que las partes interesadas en el proyecto están bien informadas del estado de las pruebas.

Entre las ventajas de la supervisión de pruebas se incluyen:

- **Mayor calidad del software:** La supervisión de pruebas, al ofrecer una visibilidad inmediata de la ejecución de las pruebas, permite a los equipos detectar y abordar los defectos en una fase temprana del SDLC, lo que se traduce en lanzamientos de software de calidad superior.
- **Mayor rapidez de comercialización:** La supervisión constante facilita la rápida identificación y resolución de problemas, lo que reduce los ciclos de desarrollo y acelera la comercialización.
- **Mayor eficacia:** La supervisión de pruebas automatiza las tareas manuales relacionadas con la ejecución y el análisis de pruebas, liberando un tiempo valioso para que los probadores se concentren en actividades más avanzadas.
- **Mejora de la toma de decisiones:** La información derivada de la supervisión de pruebas, basada en datos, permite a los equipos tomar decisiones bien fundadas sobre la asignación de recursos, la estrategia de pruebas y la planificación de lanzamientos.

Control de pruebas

Mientras que la monitorización de pruebas ofrece una perspectiva clara de las tareas de pruebas en curso, el control de pruebas es un aspecto crucial de las pruebas de software que ayuda a los equipos a tomar medidas correctivas basadas en las observaciones obtenidas de la supervisión de pruebas. En pocas palabras, es la práctica de gestionar y regular activamente el proceso de pruebas para garantizar que se ajusta a los objetivos del proyecto, es eficaz y ofrece resultados fiables.

Esta fase implica tomar decisiones basadas en la información obtenida del proceso de monitorización de las pruebas. Aquí se priorizarán las pruebas, se revisarán los calendarios de pruebas, se introducirán cambios en el entorno de pruebas y se podrán hacer otros ajustes relacionados con las actividades de pruebas para mejorar la eficacia y la calidad del futuro proceso de pruebas.

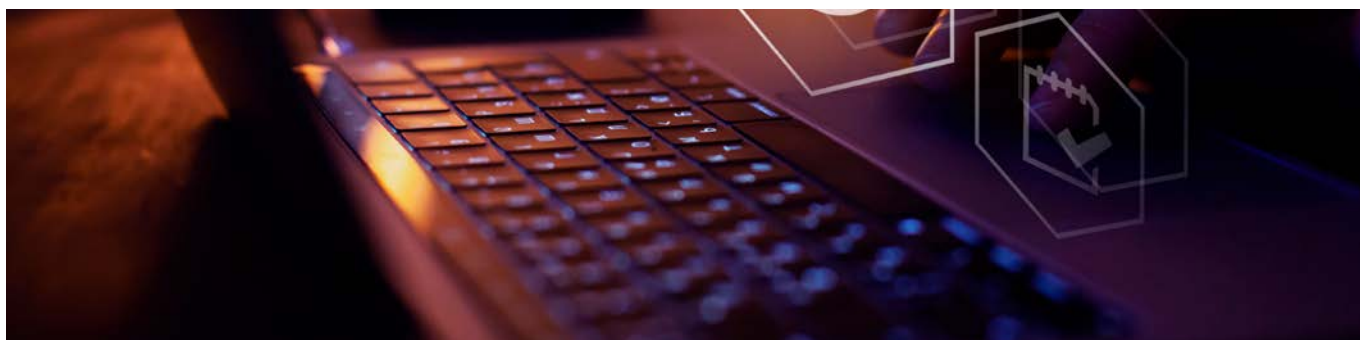
Implica:

- Establecer directrices y procesos que proporcionen un marco estructurado para las pruebas.
- Fijar normas que dicten el proceso de pruebas y ayuden a garantizar la coherencia y el cumplimiento de las mejores prácticas.
- Decidir la distribución de recursos, incluyendo personal, herramientas y entornos, para facilitar y reforzar las actividades de pruebas.
- Observar el progreso de las actividades de pruebas, lo que incluye la ejecución de las pruebas, la cobertura, la identificación y la resolución de defectos.
- Supervisar las métricas clave para evaluar la eficacia de las pruebas y yuxtaponerlas a criterios predefinidos.
- Mejorar el proceso de pruebas para garantizar la eficiencia y la eficacia en la entrega de un producto de primera categoría.

4.3 Análisis de pruebas

El análisis de prueba es una fase crítica que se centra en revisar y evaluar los requisitos y especificaciones del software para identificar condiciones de prueba y diseñar casos de prueba efectivos. Durante esta etapa, se examinan detalladamente los documentos de requisitos para comprender completamente lo que el software debe hacer, identificando tanto las funcionalidades explícitas como las implícitas.

El propósito principal del análisis de pruebas es garantizar que todas las funcionalidades y comportamientos del software sean evaluados. Esto implica identificar las condiciones de prueba, que son las situaciones y escenarios bajo los que se evaluará el software. Estas condiciones se priorizan según su importancia y el riesgo asociado, permitiendo enfocarse primero en las áreas más críticas del software.



4.4 Diseño de pruebas

En la fase de diseño de pruebas se procede a la creación de casos de prueba y conjuntos de datos de prueba basados en las condiciones de prueba identificadas durante el análisis de pruebas. En esta fase, se transforman los requisitos y especificaciones en casos de prueba específicos y ejecutables, asegurando que se cubran todas las posibles situaciones y escenarios que el software pueda enfrentar (o al menos las más importantes, como se ha explicado en el apartado Técnicas de diseño de pruebas)

El diseño de pruebas se enfoca en desarrollar casos de prueba que describen los pasos necesarios para ejecutar cada prueba, así como los resultados esperados. Estos casos de prueba deben ser claros, concisos y fácilmente entendibles para que cualquier miembro del equipo pueda ejecutarlos sin ambigüedades. La claridad y el detalle son cruciales, ya que facilitan la identificación de defectos y aseguran que las pruebas se realicen de manera consistente y reproducible.

Se pueden definir los casos de prueba a distintos niveles de abstracción o detalle. Entre otras formas de especificación de casos de prueba, tenemos:

Especificación en texto libre

Sin estructura predeterminada, se describe en un párrafo el caso de prueba y los resultados esperados. Son rápidos de implementar, pero a veces pueden dar lugar a errores por malentendidos, suelen usarse para pruebas exploratorias o tests no formales.

Paso a paso

Se escribe cada acción del usuario, con los datos necesarios para realizarlo y el resultado esperado. Tiene la ventaja de ser muy claro y fácil de seguir, incluso para quienes no conozcan la funcionalidad, por otra parte, requieren más tiempo en la definición y son más difíciles de mantener.

Especificación en lenguajes BDD, como Gherkin

Aunque están a un nivel más alto, menos detallado, proveen la suficiente información como para ejecutar los tests. Tienen varias ventajas, entre ellas, la proximidad a los lenguajes de programación de tests automáticos, y que, al seguir una estructura fija, aseguran la homogeneidad y exactitud.

Además, durante el diseño de pruebas, se crean y preparan los datos de prueba necesarios. Estos datos son fundamentales para simular las condiciones reales en las que el software operará.

4.5 Implementación de pruebas

La implementación de pruebas se centra en preparar el entorno necesario para la ejecución de los casos de prueba diseñados. Esta etapa implica configurar el hardware y software requeridos, instalar y configurar herramientas de prueba, y cargar los datos de prueba pertinentes. La correcta implementación de pruebas asegura que el entorno de prueba se asemeje lo más posible al entorno de producción, lo cual es vital para obtener resultados precisos y fiables.

Durante la implementación de pruebas, se establece el entorno de pruebas, incluyendo la configuración de servidores, bases de datos, redes y cualquier otro componente técnico necesario para soportar las pruebas. Además, se instalan y configuran las herramientas de pruebas automatizadas si se utilizan, asegurando que estén listas para ejecutar los casos de prueba de manera eficiente.

Otro aspecto importante de la implementación de pruebas es la preparación y carga de los datos de prueba. Los datos de prueba deben estar cuidadosamente seleccionados y configurados para cubrir todos los escenarios de prueba identificados durante el análisis y diseño de pruebas. Esto incluye datos normales, extremos y de borde que permitan evaluar el comportamiento del software bajo diversas condiciones.

La implementación de pruebas también puede involucrar la creación de scripts automatizados para agilizar la ejecución de casos de prueba repetitivos y reducir el esfuerzo manual. Estos scripts deben validarse para asegurar el buen funcionamiento y cubrir los casos de prueba adecuadamente.

4.6 Ejecución de pruebas

Durante la ejecución de pruebas se llevan a cabo los casos de prueba previamente diseñados y preparados: los testers ejecutan manualmente o a través de herramientas automatizadas los pasos definidos en los casos de prueba, observando y documentando los resultados obtenidos en comparación con los resultados esperados.

La ejecución de pruebas implica la implementación práctica de los escenarios de prueba en el entorno configurado durante la fase de implementación. Los testers siguen meticulosamente los casos de prueba para asegurar que todas las funcionalidades del software sean evaluadas bajo diversas condiciones y situaciones. Esto incluye pruebas funcionales, de rendimiento, de seguridad y otras que sean relevantes para el proyecto.

Durante la ejecución, los testers documentan cualquier discrepancia o defecto que encuentren. Estos defectos se registran en sistemas de gestión de errores, donde se detalla la naturaleza del problema, los pasos para reproducirlo, el entorno en el que ocurrió y su impacto en el software. Esta documentación es crucial para que los desarrolladores puedan comprender y corregir los problemas de manera efectiva.

Además, la ejecución de pruebas no es un proceso lineal; a menudo requiere retroalimentación y ajustes continuos. Si se identifican problemas críticos, puede ser necesario realizar pruebas de regresión para asegurar que las correcciones no introduzcan nuevos errores en áreas previamente funcionales del software.

La ejecución de pruebas también implica la generación de informes de prueba que resumen los resultados obtenidos, incluyendo la cobertura de pruebas, el número de defectos encontrados, su gravedad y el estado de cada caso de prueba. Estos informes proporcionan una visión clara del estado del software y ayudan a tomar decisiones informadas sobre su liberación.

4.7 Cierre de pruebas

El cierre de pruebas es la fase final del ciclo de vida de pruebas, donde se completan y documentan todas las actividades relacionadas con las pruebas del software. En esta etapa, se asegura que todos los casos de prueba se han ejecutado y resuelto todos los defectos críticos. El objetivo principal es garantizar que el software esté listo para ser liberado o puesto en producción.

El cierre de pruebas implica la revisión de todos los resultados de las pruebas para confirmar que se han cumplido los objetivos establecidos en el plan de pruebas. Se verifica que todos los requisitos han sido probados y que los resultados obtenidos cumplen con los criterios de aceptación definidos. Este proceso asegura que no se omita ninguna funcionalidad crítica y que el software sea de alta calidad y fiable.

También se realiza la documentación de los resultados de las pruebas, incluyendo los informes finales que resumen la cobertura de las pruebas, los defectos encontrados y su estado, y cualquier problema pendiente que aún deba ser resuelto. Estos informes proporcionan una visión clara y concisa del estado del software y ayudan a las partes interesadas a tomar decisiones informadas sobre su liberación.

Parte del cierre de pruebas también incluye la revisión y documentación de las lecciones aprendidas y las mejores prácticas. Esto implica analizar lo que funcionó bien y lo que no, y cómo se pueden mejorar los procesos de prueba en futuros proyectos. Esta retroalimentación es vital para mejorar continuamente la eficiencia y efectividad del proceso de pruebas.

Finalmente, se planifican y documentan las actividades de mantenimiento y pruebas de regresión futuras. Esto asegura que el software siga siendo probado y mantenido adecuadamente después de su liberación, especialmente si se realizan cambios o se añaden nuevas funcionalidades.

5 Herramientas de soporte para el proceso de QA Testing



5.1 Herramientas de gestión de pruebas

Las herramientas de gestión facilitan la gestión del ciclo de vida del desarrollo de software (SDLC) a todo el equipo, controlando la calidad en los requisitos, permitiendo que el equipo de QA pueda organizarse y administrar los flujos de pruebas, o sea, el diseño de los casos de prueba, la gestión de datos de prueba asociados, la ejecución de casos, el registro de resultados, la comunicación de defectos detectados, las métricas, etc.

Entre el ecosistema de herramientas existentes para este fin, podemos destacar algunas:

Jira: Xray

Jira es una herramienta de seguimiento de problemas y gestión de proyectos que, combinada con el plugin Xray, se convierte en una poderosa herramienta de gestión de pruebas. Permite a los testers documentar y gestionar sus hallazgos de manera eficaz.

Entre sus características se incluyen:

Seguimiento de problemas y defectos

Integración con plugins de gestión de pruebas como Xray

Gestión de proyectos y tareas

Personalización y automatización de flujos de trabajo

TestRail

Es una solución de gestión de casos de prueba para aseguramiento de la calidad y equipos de desarrollo, que está diseñada para ayudar a los usuarios a organizar, gestionar y rastrear el proceso de prueba de software de una empresa.

Entre sus características se incluyen:

Informes y métricas en tiempo real

Múltiples opciones de gestión de detalles de casos

Organización de casos en paquetes y jerarquías

Generación de reportes a medida

Integración con otras herramientas de rastreo de errores

Integración con otras herramientas de CI/CD

OpenText Application Quality Management (ALM)

Funciona como un panel único para la gestión completa de la calidad de software, ayuda a la implantación de procesos de ciclo de vida controlados. Se puede instalar en local o usarse como SaaS.

Sus funciones principales son:

Gestión del ciclo de vida: versiones, ciclos, cuadros de mandos, informes, integración con Excel

Gestión de requisitos: versionado de especificaciones, cuadros de mando, flujos adaptables

Gestión de pruebas: planes de prueba y programación de la ejecución, integración con Opentext Funcional Testing, virtualización de los servicios, configuraciones de entornos, gestión de recursos de pruebas

Gestión de defectos, asignación, trazabilidad hasta el código fuente y la compilación

OpenText Software Delivery Management (ALM Octane)

Es un software de gestión comercializado por Opentext que aglutina planificación integrada, integración continua, gestión de pruebas y gestión de versiones, aportando a los proyectos trazabilidad y visibilidad para el análisis de KPIs y mejora de la calidad continua. Permite lanzamiento de pruebas automáticas totales o parciales como parte de la integración continua. En cuanto a gestión de proyecto ofrece tableros ágiles, planificación de backlogs, sprints y versiones, y permite el escalado de agile (SAFe); para las pruebas pueden realizarse de manera manual, integrarse con herramientas de automatización y alinear los flujos con los requisitos del negocio. Analiza los cambios en el código para identificar riesgos, permite definir alertas sobre posibles desviaciones y establece métricas que le permitirán calcular los costes de cada fase y el ROI. Se integra con las herramientas DevOps más conocidas (Git, Jenkins y Jira), pruebas de seguridad, cobertura del código y vulnerabilidades en su ciclo de desarrollo. y con herramientas de chat como Teams. Genera informes detallados de los indicadores clave, cuadros de mando y permite compartílos. Se instala en los servidores de la empresa cliente.

TestMo

TestMo es una herramienta de gestión de pruebas que permite unificar en un solo punto las especificaciones, planes, ejecuciones y resultados de prueba. Los ítems de casos de test pueden ser de dos tipos predefinidos: paso a paso o texto libre. Ambos tipos se pueden personalizar añadiendo nuevos campos, y también se pueden definir nuevos tipos de casos de test a medida del proyecto. Los casos de test se organizan en el repositorio en una estructura de carpetas similar a los sistemas de ficheros, y admiten etiquetas para su clasificación. Puede incorporar resultados de test de pruebas automáticas mediante integraciones con Selenium, Cypress, Cucumber, Appium, Playwright, entre otros, para aprovechar toda la funcionalidad de creación de informes y seguimiento de las pruebas. No tiene gestión de requisitos ni de defectos, pero tiene un alto nivel de integración con herramientas como Jira y GitLab, pudiendo crear tickets directamente en estas. Se ofrece como SaaS con un modelo de licencias basado en el número de usuarios del sistema.

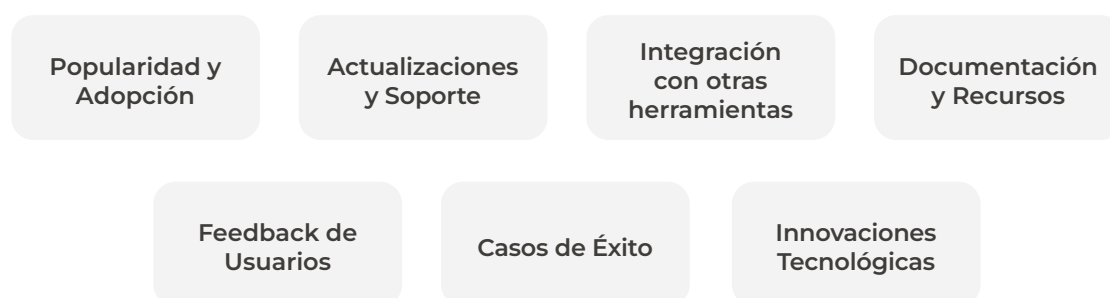
5.2 Herramientas de automatización de pruebas

Tanto para aquellos que tienen experiencia en el ámbito de la automatización como para otros que están iniciando en este mundo, seleccionar la herramienta adecuada es fundamental y compleja, os recomendamos tener en cuenta algunos criterios base según la naturaleza del proyecto, entre ellos:

Tiene facilidad de uso o requiere capacitaciones.

- Soporte para varios tipos de pruebas, incluidas las funcionales, de gestión de pruebas, móviles, etc.
- Facilidad de integración con herramientas de gestión de pruebas e incidencias (por ejemplo: Jira, Asana, AzureDevops, etc).
- Soporte para múltiples navegadores.
- Soporte para integración continua y entornos de prueba local.
- Identificación y mapeo de objetos.
- Prueba de imágenes, base de datos y recuperación de errores.
- Soporte para línea de comandos utilizado.
- Soporte para múltiples framework de automatización.
- Fácil de crear y depurar scripts de pruebas.
- Soporte para pruebas a nivel de Back-end o Front-End.
- Generación de Informes de ejecución de pruebas y resultados.
- Si es de código abierto, o requiere el uso de licencia.

Entre otros indicadores, también conviene saber:



Antes de abordar algunas herramientas de automatización, queremos recordar que no es la intención de esta sección recomendar el uso de una herramienta en particular, ya que depende en gran parte de las necesidades de cada proyecto, no obstante, hemos preparado una lista y breve reseña que os servirá de ayuda. Recuerda que puedes profundizar con más detenimiento accediendo al sitio oficial de la herramienta:

Selenium

Son un conjunto de herramientas para la automatización de pruebas de aplicaciones web. Soporta múltiples navegadores y sistemas operativos, y se integra con varios lenguajes de programación como Java, C# y Python. Proporciona alta flexibilidad y extensibilidad para pruebas funcionales de aplicaciones web, permitiendo a los testers automatizar tareas repetitivas y garantizar la calidad del software de manera eficiente.

Cypress

Es una herramienta de código abierto (con un modelo de pago para servicios especiales, pero la versión gratuita es bastante completa), destaca por su capacidad de realizar pruebas de front-end en tiempo real, proporciona una curva de aprendizaje sencilla para implementar pruebas funcionales sin necesidad de tener un dominio del framework al 100%, gracias a una interfaz intuitiva, con complementos y herramientas cliente que permiten habilitar un navegador y ejecutar las pruebas, una vez completadas, se puede revisar las instantáneas del DOM y tener la capacidad de regresar y ver el estado en cada paso, por lo que es ideal para llevar a cabo pruebas de integración y E2E con mucha practicidad. Desde la instalación inicial hasta la implementación de ejecuciones en entornos CI/CD es bastante flexible, ideal para integrarse con frameworks y herramientas modernas como React y Angular lo que la hace ideal para evaluar aplicaciones web modernas, su sitio web cuenta con buena documentación, recursos y soporte, no tendrás inconvenientes para valorar el potencial de esta herramienta en tu proyecto.

Playwright

Se utiliza comúnmente para automatizar pruebas de interfaz de usuario web, ya que ofrece una gran cobertura de navegadores, lo cual facilita realizar las pruebas funcionales, de regresión y end-to-end, garantizando que la aplicación web funcione correctamente en diferentes navegadores y dispositivos. Aunque se enfoca principalmente en el frontend, sus capacidades de automatización también permiten que se integre con otras herramientas para un flujo más completo. Es bastante intuitiva de usar, sobre todo para la creación de pruebas mediante su generador, el cual permite interactuar con la web como un usuario real, abrir la URL, navegar haciendo clics y generar las condiciones de aceptación (expects) que desees validar de forma sencilla. La ejecución de pruebas es más rápida en comparación con otras herramientas, su configuración es adaptable, por ejemplo: tiene la capacidad de capturar

automáticamente capturas de pantalla y videos, y se puede configurar para que solo capture y descargue videos si encuentra algún fallo en la ejecución. En referencia a la documentación puede mejorar, aunque es fácil de implementar hay aspectos donde se valorarían más detalles (en comparación con otras herramientas).

UIPath

UIPath, aunque originalmente diseñado para la automatización de procesos de negocio (RPA), se ha consolidado como una herramienta efectiva para pruebas funcionales de calidad gracias a su capacidad de automatizar tareas repetitivas, su facilidad de uso y su flexibilidad para cubrir diferentes tipos de pruebas. Si bien no es una herramienta especializada exclusivamente en QA, su integración con otras herramientas (Jira, ALM o Azure DevOps) y su adaptabilidad hacen que sea una opción atractiva en muchos escenarios de pruebas funcionales. UIPath permite automatizar interacciones con aplicaciones y sistemas, simulando la acción de un usuario real (como hacer clic, ingresar datos, validar resultados). Esto es esencial en pruebas funcionales, donde es necesario asegurarse de que las funcionalidades del sistema se comporten correctamente.

OpenText Functional Testing (anteriormente UFT)

Es una herramienta de automatización de pruebas que utiliza inteligencia artificial para probar aplicaciones web, móviles, de escritorio y mainframe. Permite centralizar y automatizar pruebas funcionales, de regresión y end-to-end en distintos entornos, abarcando desde la interfaz de usuario hasta las API, lo que contribuye a mejorar la calidad del software con un enfoque versátil y escalable. Su motor de IA facilita la creación y mantenimiento de pruebas mediante el reconocimiento avanzado de objetos y la posibilidad de utilizar scripts en lenguaje natural. Además, soporta la ejecución paralela de pruebas en múltiples entornos, lo que reduce significativamente los tiempos de prueba y mejora la cobertura. Es capaz de probar una amplia gama de aplicaciones (incluido SAP GUI, HANA y Fiori), autocorregir las pruebas que fallan y generar automáticamente datos de prueba. La herramienta se integra con procesos de integración y entrega continua (CI/CD). Además, OpenText permite ampliar la funcionalidad de Functional Testing mediante integración con otras herramientas propias para pruebas en móviles, de rendimiento y de gestión del ciclo de vida.

Appium

Sigue siendo la opción principal para automatizar pruebas en aplicaciones móviles en múltiples plataformas (iOS, Android, Tizen), nativas, web e híbridas, usando un solo código base lo cual facilita el mantenimiento y reduce el esfuerzo de desarrollo. Permite ejecutar pruebas en dispositivos reales y emuladores/simuladores, proporcionando flexibilidad y mayor cobertura en las pruebas. También ofrece compatibilidad con diversos lenguajes de programa-

ción como Java, JavaScript, Python, y Ruby. Se integra fácilmente en flujos de (CI/CD) como Jenkins y Bamboo, fundamental para mantener un ciclo de desarrollo ágil. Cuenta con una amplia comunidad lo cual ofrece abundante documentación, recursos y soporte, esencial para resolver problemas y optimizar las prácticas de automatización.

WebdriverIO

Es una herramienta cada vez más popular en el ámbito de la automatización de servicios API como de interfaz de usuario, lo que permite ampliar su eficiencia y cobertura de pruebas. También incluye plugins y servicios que amplían sus funcionalidades, como capturas de pantalla, reporte de errores y ejecución en paralelo. Soporta múltiples navegadores y se integra con una amplia gama de frameworks de prueba como, Selenium, Appium, Puppeteer, Mocha, Jasmine y Cucumber. Es una herramienta ideal para pruebas end-to-end (E2E), permitiendo simular flujos de usuario completos. La documentación en su sitio web es bastante buena, además, hay numerosos recursos y ejemplos disponibles en línea que puedes usar de referencia para llevar a cabo tus pruebas de concepto, sin duda una opción muy potente que debes probar si buscas una herramienta que ofrezca versatilidad, facilidad de uso y que combine ambos enfoques de prueba (frontend/backend), todo en un mismo framework.

Karate DSL

Ha ganado reconocimiento por su enfoque simplificado y eficiente para la automatización de pruebas de API e interfaz de usuario. Su capacidad para escribir pruebas en Lenguaje Natural, usando archivos feature con sintaxis similar a Gherkin, lo cual facilita las tareas de mantenimiento y su comprensión para desarrolladores y probadores sin experiencia en lenguajes de programación. Soporta funciones avanzadas para realizar pruebas de servicios REST y SOAP, manejo de datos, validaciones complejas, permitiendo la creación de “mocks” de servicios, lo cual aumenta la cobertura con poca codificación y flexibilidad. Es una herramienta que se integra fácilmente con CI/CD pipelines y otros frameworks como JUnit y Maven. La documentación en su sitio web es muy completa, por defecto después de su instalación también podemos acceder a una serie de “scripts” de prueba a modo demostrativo para evidenciar muchas de sus características funcionales y otras utilidades en pocos pasos. Si lo que buscas es una herramienta que combine en un solo marco de trabajo las pruebas de backend y frontend, esta puede ser una opción bastante competitiva.

JUnit

Marco de pruebas de código abierto para Java. Soporta la creación y ejecución de pruebas unitarias automatizadas, y se integra con herramientas de CI/CD, así como con entornos de desarrollo. Juega un rol crucial en el desarrollo orientado a pruebas o TDD. Promueve el enfoque “pruebas primero, código después”, que enfatiza primero preparar las pruebas y datos para una unidad de código, para poder probar primero y desarrollar luego, en un ciclo que incluye la refactorización del código, hasta lograr el resultado esperado con el mejor código posible.

TestNG

Framework de pruebas inspirado en JUnit, pero con funcionalidades adicionales. Soporta pruebas paralelas, configuración flexible y generación de informes avanzados, y cubre diversos tipos de pruebas como unitarias, funcionales, end-to-end, integración y más.

UserTesting

Es una plataforma que permite a las empresas obtener comentarios de usuarios reales sobre la usabilidad de sus aplicaciones. Ofrece pruebas de usabilidad en línea, grabación de sesiones e informes detallados. Proporciona información valiosa sobre la experiencia del usuario y ayuda a identificar problemas de usabilidad, mejorando así la satisfacción del usuario final.

En resumen, podemos concluir:

- La automatización de pruebas utiliza herramientas especializadas de software para ejecutar automáticamente un conjunto de casos de prueba, ampliar la cobertura y acelerar la ejecución en el proceso de pruebas de software.
- El proceso de prueba y el equipo son factores importantes para que la estrategia de automatización tenga éxito.
- La selección de la herramienta de prueba depende de la tecnología utilizada para desarrollar la aplicación a probar, primero identifica los requisitos, explora varias herramientas y sus capacidades, establece las prioridades y expectativas y opta por hacer pruebas de concepto.
- Incorpora las herramientas adecuadas para cada necesidad, a menos que haya una que cumpla con las expectativas.

- El enfoque de ejecución se debe centrar en verificar de manera eficiente si la inclusión de nuevos desarrollos o funcionalidades en el software están funcionando según lo esperado o no.
- Las pruebas manuales y de automatización se complementan entre sí para asegurar la calidad del desarrollo de software, no todo se puede automatizar.
- Es una estrategia clave para complementar los procesos de integración y entrega continua (CI/CD).

5.3 Herramientas de monitoreo y análisis

Las herramientas de monitorización y análisis son sistemas informáticos utilizados para rastrear y supervisar en tiempo real el rendimiento y la funcionalidad de redes, infraestructuras, sistemas y diferentes tipos de aplicaciones. Proporcionan información crítica que ayuda a identificar y resolver problemas antes de que impacten la productividad o la seguridad.

Estas plataformas se conectan a los diversos sistemas de la organización para recopilar datos y métricas. Para facilitar la interpretación de estos datos, las presentaciones son muy visuales, permitiendo obtener una visión completa del estado de los sistemas de un solo vistazo y detectar cualquier anomalía o problema potencial.

A continuación, se mencionan algunas de las herramientas de monitoreo y análisis con las que nos podemos encontrar:

Dynatrace

Es una herramienta de monitoreo de aplicaciones que proporciona supervisión completa y análisis en tiempo real del rendimiento y la experiencia del usuario. Ofrece funcionalidades específicas para detectar y diagnosticar problemas en aplicaciones y microservicios, facilitando la identificación de cuellos de botella y la optimización del rendimiento.

Entre sus características se incluyen:

Monitoreo de rendimiento
en tiempo real

Detección automática de
problemas y diagnósticos

Análisis de experiencia
del usuario

Integración con herramientas de
desarrollo y gestión de proyectos

New Relic

Es una herramienta de monitoreo de aplicaciones que proporciona supervisión completa y análisis en tiempo real del rendimiento y la experiencia del usuario. Ofrece funcionalidades específicas para detectar y diagnosticar problemas en aplicaciones y microservicios, facilitando la identificación de cuellos de botella y la optimización del rendimiento.

Entre sus características se incluyen:

Monitoreo de aplicaciones y servidores en tiempo real

Detección y diagnóstico de problemas

Análisis de rendimiento y experiencia del usuario

Integración con herramientas de desarrollo y gestión de proyectos

AppDynamics

Es una solución de monitoreo de aplicaciones que ayuda a las empresas a gestionar el rendimiento de sus aplicaciones y a garantizar una experiencia óptima para los usuarios. Ofrece herramientas para supervisar el rendimiento, identificar problemas y optimizar el rendimiento de las aplicaciones.

Entre sus características se incluyen:

Monitoreo de rendimiento de aplicaciones en tiempo real

Detección de problemas

Análisis de experiencia del usuario

Integración con herramientas de desarrollo y gestión de proyectos

Datadog

Es una plataforma de monitoreo y análisis de datos que proporciona una visión unificada del rendimiento de las aplicaciones, la infraestructura y los logs. Permite a los equipos de desarrollo y operaciones supervisar y optimizar el rendimiento de sus sistemas en tiempo real.

Entre sus características se incluyen:

Monitoreo de aplicaciones,
infraestructura y logs en tiempo real

Detección y diagnóstico
de problemas

Análisis de rendimiento y experiencia
del usuario

Integración con herramientas de
desarrollo y gestión de proyectos

5.4 Herramientas de rendimiento

Estas herramientas sirven para probar el comportamiento del sistema al someterse a distintos niveles de estrés o carga, verificar el tiempo de respuesta y otros parámetros para determinar si el sistema responde dentro de los parámetros marcados y si el rendimiento del software es óptimo o recomendar adecuaciones al producto.

k6

k6 es desarrollado por Grafana Labs y la comunidad. k6 es una herramienta de código abierto que hace que las pruebas de rendimiento sean fáciles y productivas para los equipos de ingeniería. k6 es gratuito, centrado en el desarrollador y extensible. Las características clave incluyen: herramienta CLI con APIs fáciles de usar para desarrolladores, scripting en JavaScript ES2015/ES6 (con soporte para módulos locales y remotos) y “checks” y “thresholds”, para pruebas de rendimiento orientadas a objetivos y fáciles de automatizar. k6 está optimizado para un consumo mínimo de recursos y diseñado para ejecutar pruebas de alta carga. Además, con k6, se puede automatizar y programar para lanzar pruebas con mucha frecuencia con una carga pequeña para validar continuamente el rendimiento y la disponibilidad del entorno de Producción.

JMeter

Es una herramienta de Apache para pruebas de carga y rendimiento que soporta aplicaciones web, bases de datos y servicios web, permitiendo la simulación de cargas pesadas. Es ideal para pruebas de rendimiento y carga, proporcionando una interfaz fácil de usar y capacidades avanzadas de scripting. Permite a los testers evaluar cómo se comporta el sistema bajo diferentes niveles de carga y encontrar cuellos de botella.

Gatling

Esta herramienta de código abierto ofrece una solución de “pruebas de carga y rendimiento como código”. Es decir, permite almacenar los scripts de prueba como código, lo que implica tener un histórico de cambios y versiones. Permite evitar caídas anticipando los fallos y los tiempos de respuesta lentos, detectar los problemas con antelación para mejorar el tiempo de comercialización, mejorar la experiencia del usuario, etc. Está pensado para pruebas de carga continuas, integradas con los flujos de CI/CD. Tiene un lenguaje específico de dominio fácilmente comprensible.

OpenText Performance Engineering (anteriormente LoadRunner)

Perteneciente a OpenText, LoadRunner es una potente suite propietaria de herramientas tanto on-premise como cloud. Ofrece en su propio ecosistema todo lo necesario para las pruebas de rendimiento: grabación y desarrollo de scripts, creación y ejecución de escenarios, gestión de inyectores, monitorización online y offline, trazabilidad de pruebas, etc. Aparte del estándar HTTP, permite realizar pruebas de rendimiento para un amplio abanico de tecnologías: interfaz web, Citrix, SAP, RDP, Java, .NET, bases de datos, etc. Además, puede ejecutar scripts creados en otras herramientas como JMeter, Gatling, NUnit y Selenium. A partir de un número de usuarios virtuales tiene coste de licenciamiento.

Locust

Otra herramienta de pruebas de carga de código abierto. Permite definir el comportamiento de usuarios de prueba mediante código Python, lo que lo convierte en una herramienta muy potente, aunque a costa de que el tester debe tener cierto conocimiento técnico o trabajar en conjunto con el equipo de desarrollo para definir las pruebas. Es una herramienta altamente distribuida y escalable que admite la ejecución de pruebas distribuidas en varias máquinas y simular un número muy alto de usuarios con facilidad.

5.5 Herramientas de calidad de código

Estas herramientas analizan el código sin ejecutarlo, identificando posibles errores y áreas de mejora, vulnerabilidades, bugs, code smells (mala práctica), deuda técnica, cobertura de código en las pruebas, entre otras métricas. Las herramientas de calidad de código permiten la detección temprana de defectos antes de realizar las pruebas dinámicas. Algunas de estas herramientas comúnmente usadas en la industria del software son:

SonarQube

Esta herramienta para el análisis continuo de la calidad proporciona información sobre vulnerabilidades, code smells, cobertura de código, y más. Soporta múltiples lenguajes de programación y es ideal para equipos que desean integrar análisis de calidad en su pipeline (tubería) de CI/CD.

Coverage.py

Herramienta para medir la cobertura del código en programas escritos en Python que proporciona informes detallados sobre qué partes del código han sido ejecutadas, resalta las líneas no cubiertas y genera informes HTML.

JaCoCo

Herramienta de cobertura de código para aplicaciones Java que soporta integración con herramientas de construcción como Maven y Gradle, y puede generar informes detallados de cobertura en varios formatos.

Istanbul

Herramienta de cobertura de código para aplicaciones JavaScript que proporciona informes de cobertura detallados y se integra bien con marcos de prueba como Mocha y Jasmine.

Anchore

Herramienta de análisis estático basado en políticas que automatiza la inspección, análisis y evaluación de imágenes Docker para asegurar los despliegues garantizando que el contenido cumple ciertos criterios. Notifica a los usuarios los resultados obtenidos y las vulnerabilidades descubiertas.

Snyk

Plataforma de pruebas y seguridad de aplicaciones. Diseñada para descubrir vulnerabilidades en el código, contenedores y bibliotecas de código abierto. Ofrece supervisión en tiempo de ejecución, alertas y notificaciones. Permite verificar proyectos directamente desde el repositorio mediante la integración con los flujos de CI/CD, así como agregar pruebas automatizadas.

5.6 Herramientas de seguridad

Se trata de aplicaciones y servicios diseñados para identificar, analizar y corregir vulnerabilidades en el software.

Estas herramientas aseguran que el producto cumpla con los estándares de seguridad y se proteja contra posibles ataques o fallos, pueden incluir:

- **Escáneres de vulnerabilidades:** Detectan debilidades en el código y configuraciones que podrían ser explotadas.
- **Análisis de código estático (SAST):** Revisa el código fuente para identificar vulnerabilidades antes de la ejecución.
- **Análisis de código dinámico (DAST):** Examina la aplicación en ejecución para descubrir problemas de seguridad.
- **Pruebas de penetración (Pen Testing):** Simulan ataques reales para identificar y corregir fallos de seguridad.

Estas herramientas son esenciales para garantizar que el software sea seguro y resistente a amenazas antes de su lanzamiento.

A continuación, se mencionan algunas de las herramientas de seguridad con las que nos podemos encontrar:

Nessus

Es una herramienta de análisis de vulnerabilidades que permite realizar evaluaciones de seguridad en redes y sistemas. Ofrece funcionalidades específicas para identificar y gestionar vulnerabilidades, facilitando la protección de los activos de TI contra amenazas conocidas.

Entre sus características se incluyen:

Escaneo de vulnerabilidades
en redes y sistemas

Generación de informes
detallados de seguridad

Identificación y gestión de
vulnerabilidades

Integración con herramientas de
gestión de incidentes y seguridad

Burp Suite

Es una herramienta de pruebas de seguridad para aplicaciones web que permite realizar pruebas de penetración y análisis de seguridad. Ofrece funcionalidades específicas para detectar vulnerabilidades en aplicaciones web, facilitando la identificación y explotación de fallos de seguridad.

Entre sus características se incluyen:

Pruebas de penetración en
aplicaciones web

Generación de informes
detallados de seguridad

Análisis de vulnerabilidades de
seguridad

Integración con herramientas de
desarrollo y gestión de proyectos

OpenVAS

Es un marco de análisis de vulnerabilidades que permite detectar problemas de seguridad en redes y sistemas. Ofrece funcionalidades específicas para escanear y gestionar vulnerabilidades, facilitando la protección de los activos de TI contra amenazas conocidas.

Entre sus características se incluyen:

Escaneo de vulnerabilidades
en redes y sistemas

Generación de informes
detallados de seguridad

Identificación y gestión de
vulnerabilidades

Integración con herramientas de
gestión de incidentes y seguridad

Metasploit

Es una plataforma de pruebas de penetración que permite realizar exploraciones, evaluaciones y explotación de vulnerabilidades. Ofrece funcionalidades específicas para identificar y explotar fallos de seguridad, facilitando la evaluación de la seguridad en redes y sistemas.

Entre sus características se incluyen:

Pruebas de penetración en
redes y sistemas

Generación de informes
detallados de seguridad

Identificación y explotación de
vulnerabilidades

Integración con herramientas de
gestión de incidentes y seguridad

5.7 Herramientas para evaluar accesibilidad web

Existen programas de software y servicios que ayudan a los equipos de desarrollo (DEV&QA) a identificar y corregir los problemas de accesibilidad, permiten identificar barreras de accesibilidad ahorrando tiempo y esfuerzo en pruebas, no obstante, las herramientas tienen algunas limitaciones, no todo se pueden automatizar y se deben realizar algunas comprobaciones de forma manual “asistida por la herramienta”, algunas requieren el uso de licencia (de pago), y otras son de libre uso, su elección dependerá de la necesidad del entorno web, sus y sus características.

En general, la mayoría se integran en diferentes entornos de trabajo, bien mediante el uso de librerías, hasta un complemento o extensión ejecutada desde el navegador web, algunas pueden escanear sitios buscando problemas de accesibilidad, otras solo escanean la página, pero hay que aclarar que, sin importar la herramienta usada, algunos resultados pueden ser inexactos (falsos/positivos) por lo que hay que verificarlo más. A continuación, se mencionan algunas recomendaciones de las herramientas para la evaluación de accesibilidad web y aspectos más relevantes:

Axe DevTools Extension

Es una herramienta desarrollada por el equipo de Deque Systems, Inc., se instala como complemento o extensión en el navegador para pruebas de accesibilidad, combina pruebas automatizadas como guiadas y cuenta con una interfaz fácil de usar. Su enfoque de ejecución combinada permite detectar el 80% de los problemas más comunes de accesibilidad.

El proyecto axe-core

Facilita que las pruebas de accesibilidad se pueden ejecutar como parte integral del ciclo de pruebas en CI/CD del proyecto, permitiendo encontrar un promedio del 57% de los problemas WCAG (Por sus siglas en inglés, Web Content Accessibility Guidelines que significa, Guía para la accesibilidad de contenidos web) automáticamente, el resto de los elementos que no puede identificar los devolverá como “incompletos” para aplicar revisiones manuales. También permite realizar escaneos en todo el sitio web o parciales, aprendizaje automático, generar y compartir enlaces con los resultados, guardar y exportar resultados en diferentes formatos.

Wave Evaluation Tool (Herramienta de evaluación de olas)

Es una herramienta web desarrollada por WebAIM.org se instala a través de una extensión en el navegador y proporciona información visual mediante la inyección de iconos e indicadores en la página auditada lo que la hace una opción bastante intuitiva y educa sobre problemas de accesibilidad. Como otras herramientas de ejecución automatizada ayuda a encontrar problemas de accesibilidad, pero con cierto grado de cobertura y precisión, lo cual hace necesaria la intervención humana. El análisis se realiza íntegramente dentro del navegador Chrome, lo que permite una valoración segura de las páginas web de intranet, locales, protegidas con contraseña y otras páginas web confidenciales. Para ejecutar un informe WAVE, simplemente haga clic en el ícono WAVE a la derecha de la barra de direcciones de su navegador, o seleccione “WAVE esta página” en el menú contextual. La interfaz facilita la evaluación humana de muchos otros aspectos de la accesibilidad y los problemas encontrados con esta herramienta se alinean con las pautas del estándar de WCAG 2.2. También ofrece la posibilidad de realizar una auditoría en línea, cuyo reporte también es bastante intuitivo para hacer evaluaciones rápidas, para ello se debe acceder a través del siguiente enlace: <https://wave.webaim.org/> y ejecutar la auditoría.

PageSpeed Insights

Es una herramienta de Google que ayuda a analizar el rendimiento de una página web, está disponible en pagespeed.web.dev y permite ejecutar análisis bajo demanda con solo indicar la URL del sitio web, tanto para dispositivos móviles como de escritorio. Entre las métricas generadas se incluyen una enfocada a “Accesibilidad web”, muestra las auditorías y el detalle correspondiente para ayudar a corregir los problemas encontrados.

Lighthouse

Es una herramienta de código abierto desarrollada por Google para auditar el rendimiento y la accesibilidad de las páginas web. Está incluida en el navegador de Google Chrome por lo que no es necesario instalar ninguna extensión, su ejecución es sencilla, solo se abre el Chrome y se navega a la página que deseas auditar, luego se habilita las opciones Chrome DevTools (en Windows "F12"), se configuran las opciones de auditoría (Accesibilidad), se selecciona el modo del dispositivo (Mobile/Desktop) y clic en "Analyze page load" para generar el reporte. Al finalizar la ejecución podemos ver el detalle de la auditoría realizada, los problemas encontrados y recomendaciones para ayudar a corregirlos, el enfoque de esta herramienta permite hacer auditorías más completas que "PageSpeed Insights", además de facilitar las herramientas de desarrollador del Chrome DevTools para análisis y/o correcciones propuestas al cierre de la auditoría.

El equipo de W3 ha publicado en su página web un listado de herramientas:

<https://www.w3.org/WAI/test-evaluate/tools/list/> este sitio web contiene filtros que permiten encontrar la herramienta que más se adecue a las necesidades del proyecto.

6 Métricas y KPIs esenciales en QA

6.1 Métricas vs KPIs: enfoque y propósito

Las métricas y los KPI son fundamentales en el desarrollo del software, permiten determinar la calidad, la eficiencia, el progreso y el estado en el ámbito de las pruebas de software, también ayudan tomar mejores decisiones, afianzando la estrategia y promoviendo la mejora continua en los procesos, entre otras bondades.

Un KPI (Key Performance Indicator) en QA, es una métrica clave que se utiliza para evaluar el éxito o desempeño del proceso de QA en relación con sus objetivos estratégicos. Los KPIs están específicamente alineados con metas críticas, como la tasa de defectos, la cobertura de pruebas y el tiempo de resolución de problemas, ayudando a identificar áreas de mejora y éxito de manera específica y medible.

Una métrica en QA, es una medida cuantitativa utilizada para evaluar cualquier aspecto del desempeño o funcionamiento de un proceso de pruebas, sin estar necesariamente vinculada a objetivos estratégicos. Las métricas pueden incluir datos como el número de casos de prueba ejecutados, el porcentaje de automatización de pruebas o el tiempo medio de ejecución de pruebas, proporcionando información útil para monitorear y mejorar el proceso de pruebas de manera operativa.

Métricas de calidad

Evalúan la calidad general del producto. Se enfocan en aspectos como la satisfacción del usuario, la conformidad con los requisitos, y la estabilidad del sistema.

Se mide a través de diversas técnicas, como pueden ser las encuestas de satisfacción del usuario, el análisis de defectos postproducción, y las revisiones de cumplimiento de requisitos.

Un ejemplo sería la puntuación de satisfacción del usuario (NPS (Net Promoter Score) que se mide a través de encuestas donde los usuarios califican su satisfacción en una escala del 1 al 10.

Métricas de proceso

Evalúan la eficiencia y efectividad de los procesos de desarrollo y prueba.

Se mide a través del seguimiento del tiempo de ciclo, tiempo de resolución de defectos, porcentaje de tareas completadas a tiempo, número de defectos encontrados en las fases iniciales del desarrollo.

Un ejemplo sería el tiempo promedio de resolución de defectos, que se calcula midiendo el tiempo desde que se reporta un defecto hasta que se cierra.

Métricas de cobertura

Miden el grado en que el código o los requisitos han sido cubiertos por las pruebas. Se mide a través de porcentajes de cobertura de código, cobertura de requisitos y cobertura de riesgo.

Un ejemplo sería la cobertura de código, que se mide calculando el porcentaje de líneas de código que han sido ejecutadas por los casos de prueba.

Métricas de calidad de código

Evalúan la calidad del código fuente en términos de mantenibilidad, complejidad y adherencia a estándares.

Se mide a través de la complejidad ciclomática, densidad de comentarios, y número de violaciones de estilo. Algunas de las herramientas que se suelen utilizar son SonarQube, Checkstyle, PMD.

Un ejemplo sería la complejidad ciclomática, que se mide contando el número de rutas lineales independientes a través del programa.

Métricas de defectos

Miden la cantidad y la severidad de los defectos encontrados durante todas las fases del proceso de prueba.

Se mide a través del número total de defectos, tasa de defectos abiertos/cerrados, y defectos por severidad, por módulo, etc.

Un ejemplo sería la tasa de defectos abiertos/cerrados, que se calcula dividiendo el número de defectos abiertos entre el número de defectos cerrados en un periodo de tiempo.

Métricas de casos de prueba

Evalúan la efectividad y eficiencia de los casos de prueba diseñados.

Se mide a través del número de casos de prueba ejecutados, tasa de éxito/fallo de casos de prueba, casos de prueba en cada tipo, en cada módulo, porcentaje de automatización de los casos de prueba, tiempo necesario para un ciclo de ejecución.

Un ejemplo sería la tasa de éxito de casos de prueba, que se calcula dividiendo el número de casos de prueba que pasaron entre el número total de casos de prueba ejecutados.

Métricas de testing exploratorio

Evalúan la efectividad de las pruebas exploratorias, donde los testers utilizan su experiencia y creatividad para encontrar defectos.

Se mide a través del número de sesiones de prueba exploratoria, defectos encontrados por sesión, y cobertura exploratoria.

Un ejemplo sería el número de defectos encontrados por sesión, que se calcula dividiendo el número total de defectos por el número de sesiones exploratorias realizadas.

Métricas de automatización de pruebas

Evalúan la efectividad y el rendimiento de las pruebas automatizadas.

Se mide a través del número de casos de prueba automatizados, tasa de éxito de las pruebas automatizadas, y el tiempo de ejecución de las pruebas automatizadas.

Un ejemplo sería la tasa de éxito de pruebas automatizadas, que se calcula dividiendo el número de pruebas automatizadas que pasaron entre el número total de pruebas automatizadas ejecutadas.

Métricas de performance

Evalúan el rendimiento del sistema bajo diferentes condiciones de carga y estrés.

Se mide a través del tiempo de respuesta y uso de recursos (CPU, memoria).

Algunas de las herramientas que se suelen utilizar son JMeter, LoadRunner, Gatling.

Un ejemplo sería el tiempo de respuesta, que se mide registrando el tiempo que toma para que una solicitud del usuario sea procesada y una respuesta sea recibida.

6.2 KPIs fundamentales

Cobertura de pruebas

Mide el grado en que el software ha sido cubierto por las pruebas.

Se mide a través del porcentaje de requisitos, funcionalidades o líneas de código que han sido cubiertas por las pruebas.

Algunas de las herramientas que se suelen utilizar son SonarQube, JaCoCo, Cobertura. Un ejemplo sería la cobertura de código, que se calcula dividiendo el número de líneas de código ejecutadas por los casos de prueba entre el número total de líneas de código.

Pruebas creadas

Mide el número de casos de prueba diseñados durante un periodo específico.

Se mide contando el número total de casos de prueba creados.

Un ejemplo sería el número de casos de prueba creados en un sprint, que se registra en la herramienta de gestión de pruebas.

Eficacia de los casos de prueba

Mide la capacidad de los casos de prueba para identificar defectos.

Se mide a través de la relación entre el número de defectos encontrados y el número de casos de prueba ejecutados.

Un ejemplo sería calcular la eficacia dividiendo el número de defectos encontrados por el número de casos de prueba ejecutados.

Ejecución de pruebas

Mide el número de casos de pruebas ejecutados en un periodo específico.

Se mide contando el número total de casos de prueba ejecutados.

Un ejemplo sería el número de casos de prueba ejecutados en una semana, que se registra en la herramienta de gestión de pruebas.

Cobertura de ejecución de pruebas

Mide el porcentaje de casos de prueba diseñados que han sido ejecutados.

Se mide dividiendo el número de casos de prueba ejecutados entre el total diseñado.

Un ejemplo sería la cobertura de ejecución de pruebas, que se calcula dividiendo el número de casos de prueba ejecutados entre el número total de casos de prueba diseñados en un sprint.

Cobertura de casos de prueba pasados

Mide el porcentaje de casos de prueba ejecutados que han pasado.

Se mide dividiendo el número de casos de prueba que pasaron entre el total ejecutado.

Un ejemplo sería calcular la cobertura de casos de prueba pasados dividiendo el número de casos de prueba que pasaron entre el número total de casos de prueba ejecutados en un sprint.

Cobertura de casos de prueba fallados

Mide el porcentaje de casos de prueba ejecutados que han fallado.

Se mide dividiendo el número de casos de prueba fallados entre el número total de ejecutados.

Un ejemplo sería calcular la cobertura de casos de prueba fallados dividiendo el número de casos de prueba que fallaron entre el número total de casos de prueba ejecutados en un sprint.

Porcentaje de defectos reparados

Mide el porcentaje de defectos encontrados que han sido corregidos.

Se mide dividiendo el número de defectos corregidos entre el número total de defectos reportados.

Un ejemplo sería calcular el porcentaje de defectos reparados dividiendo el número de defectos corregidos entre el número total de defectos reportados en un mes.

Porcentaje de defectos críticos

Mide el porcentaje de defectos reportados que son clasificados como críticos.

Se mide dividiendo el número de defectos críticos entre el número total de defectos reportados.

Un ejemplo sería calcular el porcentaje de defectos críticos dividiendo el número de defectos críticos entre el número total de defectos reportados en un mes.

Tasa de éxito de resolución de defectos

Mide la efectividad del equipo en resolver defectos reportados.

Se mide dividiendo el número de defectos resueltos correctamente entre el número total de defectos reportados.

Un ejemplo sería calcular la tasa de éxito de resolución de defectos dividiendo el número de defectos resueltos correctamente entre el número total de defectos reportados en un periodo determinado.

Relación de calidad

Mide la calidad del producto en términos de defectos encontrados y funcionalidad entregada.

Se mide dividiendo el número de defectos reportados entre el número de funcionalidades entregadas.

Un ejemplo sería calcular la relación de calidad dividiendo el número de defectos reportados entre el número de funcionalidades entregadas en una release.

6.3 Quality Gates o rangos de calidad

Quality Gates, son niveles mínimos que se deben alcanzar en una métrica o un KPI determinado antes de comenzar una nueva fase o dar por cerrada la presente. Funcionan como barreras de calidad que garantizan que el software cumpla con los criterios predefinidos antes de progresar, lo que ayuda a identificar y corregir problemas de manera temprana y sistemática.

Cada Quality Gate tiene criterios específicos que deben cumplirse. Estos pueden incluir métricas de código, resultados de pruebas, cobertura de pruebas, revisión de requisitos, y otros estándares de calidad.

Muchas Quality Gates están automatizadas para permitir la integración y la entrega continuas (CI/CD). Las herramientas de análisis de código y pruebas automatizadas pueden evaluar si se cumplen los criterios definidos. Se establecen en varios puntos del ciclo de vida del desarrollo, como después de la codificación, las pruebas unitarias, la integración, y antes del despliegue en producción.

Entre sus beneficios, podemos encontrar los siguientes puntos:

Mejora de la calidad

Garantizan que el software cumple con los estándares de calidad en cada etapa del desarrollo.

Transparencia y trazabilidad

Proporcionan visibilidad sobre el estado de calidad del proyecto y permiten rastrear el cumplimiento de criterios específicos.

Reducción de riesgos

Identifican y abordan problemas de calidad de manera temprana, reduciendo el riesgo de defectos en etapas posteriores.

Eficiencia en el desarrollo

Ayudan a evitar el retrabajo y a asegurar que el software defectuoso no avance a etapas posteriores, ahorrando tiempo y recursos.

7

Buenas prácticas para un QA Testing exitoso

7.1 Estrategia QA a la medida

Una estrategia de QA a la medida plantea un enfoque personalizado y específico para asegurar la calidad del software, adaptándose a las necesidades y características particulares de cada proyecto. Comienza con una comprensión profunda de los requisitos del cliente y los objetivos del negocio, priorizando los aspectos críticos del software desde el principio.

Incluye una planificación detallada de pruebas, definiendo objetivos, alcance, metodologías, recursos necesarios y criterios de aceptación. Se especifican los tipos de pruebas a realizar, como funcionales, de rendimiento, de seguridad y de usabilidad, garantizando una cobertura completa del software.

La automatización de pruebas es esencial, implementando herramientas y procesos que permitan realizar pruebas repetitivas y de regresión de manera eficiente. Integrar la QA en el pipeline de CI/CD (Integración Continua/Entrega Continua) facilita pruebas continuas y retroalimentación inmediata, permitiendo la identificación temprana de problemas.

La gestión de defectos establece un proceso claro para el seguimiento, priorización y resolución de defectos, con una comunicación efectiva entre los equipos de desarrollo y QA. La colaboración constante entre todos los equipos involucrados es crucial, fomentando una cultura de trabajo conjunto y utilizando herramientas adecuadas para mantener la alineación con los objetivos del proyecto.

La capacitación continua del equipo de QA asegura que estén al día con las últimas herramientas, tecnologías y mejores prácticas de pruebas, mejorando la competencia del equipo y contribuyendo a la mejora continua del proceso de QA. La estrategia debe ser dinámica y adaptable, incorporando revisiones post-implementación para identificar áreas de mejora y ajustar los procesos según sea necesario, basándose en métricas clave y KPIs para evaluar la efectividad y realizar ajustes basados en datos objetivos.

Cabe destacar que, como hemos mencionado al inicio, aplicar o no los puntos antes mencionados dependerá del tipo del proyecto en el que se va a trabajar. Por ejemplo, en un proyecto de corta duración, o de un tamaño reducido por poner otro ejemplo, quizás el proceso de desarrollo de un pipeline de CI/CD no es el punto al que se le debería dar tanta prioridad, ya

que puede haber otros puntos más importantes a los que se le deba dar más importancia.

En los siguientes puntos de esta sección de buenas prácticas para un buen QA testing iremos viendo más específicamente información acerca de las partes que lo componen. Como podría ser una sección específica a las diferentes estrategias que nos podemos encontrar en un proyecto hecho a medida.

Beneficios:

Los beneficios de una estrategia de QA ajustada a las necesidades del proyecto, son numerosos y tienen un impacto significativo en la calidad del software y eficiencia de los procesos de desarrollo.

Algunos de los principales beneficios incluyen:

Adaptación a requisitos específicos

Permite que el proceso de pruebas se ajuste a las necesidades y características particulares del proyecto, asegurando que se aborden los aspectos más críticos del software. Esto asegura que el producto final cumpla con los requisitos y expectativas del cliente.

Eficiencia y optimización de recursos

Al enfocarse en las áreas que más lo necesitan, se maximiza el uso eficiente del tiempo y el presupuesto, evitando pruebas innecesarias y dirigiendo los recursos hacia las áreas que tendrán un mayor impacto en la calidad del software.

Gestión de riesgos

Identificar y mitigar riesgos específicos del proyecto de manera temprana reduce la probabilidad de problemas graves en etapas posteriores. Esto ayuda a prevenir defectos críticos que podrían afectar la funcionalidad y la estabilidad del software.

Flexibilidad y adaptabilidad

Una estrategia de QA personalizada es más flexible y puede adaptarse rápidamente a cambios en los requisitos, feedback de los usuarios o nuevas prioridades del negocio, lo que permite una respuesta ágil a las necesidades del proyecto.

Mejora Continua

Fomenta la revisión constante y la mejora de los procesos de QA. La retroalimentación específica y el análisis de métricas permiten ajustar y optimizar las prácticas de pruebas continuamente.

Satisfacción del cliente

Al asegurar que el software cumple con los estándares de calidad y las expectativas del cliente, se mejora significativamente la satisfacción del cliente. Un producto final de alta calidad genera confianza y credibilidad.

Cumplimiento normativo

Para proyectos en industrias reguladas, una estrategia de QA a la medida garantiza el cumplimiento de todas las normas y regulaciones pertinentes, evitando problemas legales y asegurando la calidad del software.

Innovación y Desarrollo del Equipo

Proporciona oportunidades para la capacitación continua y el desarrollo profesional del equipo de QA, lo que mejora sus habilidades y conocimientos y contribuye a la innovación en el proceso de pruebas.

7.2 Diseño integral de planes de prueba

Generalmente se confunde el término de plan de pruebas con la acción diseñar los casos de prueba y en muchas organizaciones, al diseño del caso de prueba lo llaman plan de pruebas.

El plan de pruebas es un documento, que no siempre se suele generar en un proyecto de validación de software. Este documento es uno de los puntos más importantes en la estrategia de QA, fundamental en el desarrollo de un proyecto de software.

Un plan de pruebas debe recoger todos los aspectos relacionados con el testing, entre ellos:

- Cobertura completa de requisitos, para asegurar que todos los requisitos funcionales y no funcionales del software estén cubiertos.
- Cronograma alineado con la estrategia general de QA, el alcance del proyecto y una definición clara de objetivos en cada fase del proceso de pruebas.
- Enfoque y diversidad de pruebas, lo cual permitirá delimitar el alcance, se recomienda incluir diferentes tipos de pruebas tanto funcionales como no funcionales, por ejemplo: Pruebas unitarias, de integración, funcionales, de regresión, de aceptación, de rendimiento, y seguridad, para garantizar una evaluación integral del software en desarrollo.
- Criterios de aceptación y salida, para saber cuándo un conjunto de pruebas ha sido exitoso, y criterios de salida para determinar cuándo el software está listo para avanzar a la siguiente fase del ciclo de vida.
- Criterio de paso/fallo, suspensión y reanudación de las pruebas.
- Identificar funcionalidades “CORE” y establecer la planificación de Pruebas de Regresión, para asegurar que nuevas características o correcciones no afecten negativamente las funcionalidades existentes o la estabilidad del software.
- Gestión de datos de prueba (Test Data), para asegurar que se cuenten con datos de prueba adecuados y representativos que permitan una evaluación precisa de las funcionalidades del software.
- Entregables de las pruebas, documentación y reportes, para asegurar una adecuada preparación de los entregables al cierre del plan de pruebas, entre ellos: Release Note, documentación funcional, reportes de resultados, para sustentar decisiones y facilitar la trazabilidad, etc.
- Tareas e hitos clave de las pruebas.
- Entornos de pruebas (características esenciales).
- Asignación de roles y responsabilidades, se debe definir claramente quiénes son responsables de diseñar, ejecutar y dar mantenimiento a las actividades de prueba, incluyendo la colaboración entre desarrolladores, testers, entre otros participantes.
- Identificar las necesidades de formación en el equipo y actividades complementarias o dependencias con terceros.
- Automatización de pruebas, se debe identificar las áreas donde la automatización puede ser más eficaz para mejorar la eficiencia y reducir el tiempo de retroalimentación, sin dejar de lado las pruebas manuales donde sean necesarias.
- Identificar riesgos y plan de mitigaciones, etc.

7.3 Selección de herramientas adecuadas

Actualmente, la industria del software provee muchas herramientas para facilitar la gestión de las pruebas, lo cual puede resultar beneficioso, pero también confuso. Es por ello, que se recomienda invertir tiempo en el análisis de las herramientas disponibles, considerando su facilidad de uso, costo, integración con herramientas existentes y escalabilidad, entre otras variables. Esta evaluación le permitirá tomar la decisión que mejor se adapte a las necesidades del proyecto.

7.4 Estrategia CI & CD

Estrategia de Integración y Entrega continua (CI & CD)

Las estrategias de integración continua (CI) y entrega continua (CD) se utilizan en el desarrollo de software y complementan la metodología ágil. Sin CI y CD, los desarrolladores dividen su trabajo y ensamblan diferentes segmentos de código en una etapa avanzada del ciclo de desarrollo. Esto puede generar falta de cohesión, compatibilidad y problemas con la interacción de varios segmentos de código.

Integración Continua (CI)

Con la integración continua, el código se guarda en un repositorio central. Los desarrolladores trabajan para realizar cambios menores en el código y cargar pequeñas secciones de código al repositorio central con regularidad. La integración de la gestión de calidad en esta metodología implica realizar una serie de pruebas con cada actualización del código. Si bien probar los nuevos segmentos es esencial, las pruebas de regresión también son cruciales para evaluar cómo los cambios afectan las características principales del producto.

Entrega Continua (CD)

La entrega continua permite el lanzamiento regular de nuevas iteraciones de su producto, lo que proporciona un método rápido y eficiente para abordar errores y problemas que afectan la experiencia del usuario.

La clave es incorporar los comentarios de los usuarios en sus procesos de CI y CD para abordar rápidamente los problemas y lanzar una versión nueva y mejorada del producto. Nuevamente, tendrá que incorporar pruebas en su proceso, por ejemplo, haciendo que los evaluadores realicen pruebas de usabilidad antes de que una nueva versión principal de su producto esté disponible para los usuarios.

Beneficios de implementar CI & CD

Implementar CI/CD en un proyecto de desarrollo de software no solo mejora la eficiencia y calidad del proceso de desarrollo, también aportan numerosos beneficios, entre los que se destacan:

- **Mayor Frecuencia de Entregas**, permite realizar despliegues en producción con mayor frecuencia, asegurando que las nuevas funcionalidades, mejoras y correcciones de errores lleguen rápidamente a los usuarios.
- **Reducción del Riesgo**, al hacer integraciones y despliegues más pequeños y frecuentes, se reduce el riesgo de fallos en el sistema, ya que es más fácil identificar y corregir errores en cambios menores que en grandes despliegues.
- **Automatización de Pruebas**, el CI/CD permite la automatización de pruebas unitarias, funcionales y de integración, garantizando que cada cambio en el código es validado automáticamente antes de ser integrado en la rama principal o desplegado en producción.
- **Mejora en la Calidad del Software**, la integración y pruebas continuas ayudan a detectar defectos y problemas de calidad en una etapa temprana del ciclo de desarrollo, lo que facilita su resolución antes de que se conviertan en problemas mayores.
- **Feedback Rápido**, los desarrolladores reciben feedback de inmediato sobre los cambios en el código, lo que les permite corregir errores y mejorar la calidad del software de manera continua.
- **Mayor Colaboración y Sincronización**, fomenta la colaboración entre los miembros del equipo, ya que todos trabajan sobre la misma base de código, y las integraciones frecuentes ayudan a mantener el código sincronizado.
- **Entrega Rápida y Fiable**, ya que acelera el tiempo de comercialización, permitiendo a las empresas responder rápidamente a las necesidades del mercado y de los clientes con nuevas funcionalidades y mejoras.
- **Mejora en la Satisfacción del Cliente**, al entregar un software de mejor calidad y con mayor rapidez, se incrementa la satisfacción del cliente, ya que las necesidades y expectativas se cumplen de manera más eficiente.
- **Cultura de Mejora Continua**, fomenta una cultura de mejora continua en los equipos de desarrollo, promoviendo prácticas ágiles y DevOps que buscan optimizar todos los procesos del ciclo de vida del software.
- **Optimización de Recursos**, la automatización de tareas repetitivas y la reducción de errores manuales liberan tiempo y recursos, permitiendo que los equipos se concentren en tareas de mayor valor agregado.

7.5 Integración de BDD y TDD en el proceso de desarrollo Agile

En capítulos anteriores hemos abordado los enfoques de desarrollo TDD y BDD, sabemos que TDD nos permite ir de los requisitos a la programación del código, identificando primero las pruebas y luego iterar sobre la implementación de estas durante la fase de desarrollo. Por otra parte, el enfoque BDD nos permite identificar las pruebas a través del comportamiento esperado, las historias de usuario y sus criterios de aceptación son el marco de referencia para el diseño y codificación de la funcionalidad y las pruebas.

En este apartado, compartiremos una estrategia para integrar ambos enfoques en el marco de un desarrollo agile. Básicamente, aprovechar la visión del enfoque BDD en cuanto a la calidad esperada desde una perspectiva alto nivel de los requisitos y el aporte de TDD en la calidad esperada en el bajo nivel, desde la implementación.

Una manera de integrar ambos enfoques en el proceso Agile se basa en los fundamentos de DevOps y se resume a continuación:

- Historias de Usuario: El ciclo Agile comienza con un backlog de historias de usuario y sus criterios de aceptación respectivos.
- Aplicamos el enfoque BDD para definir las pruebas funcionales, para verificar cada criterio de aceptación de acuerdo con su comportamiento usando la sintaxis de Gherkin (en principio, estas pruebas deben fallar, dado que el código no está implementado, la idea es diseñar todos los escenarios a probar en una fase temprana).
- Al momento de iniciar el desarrollo, aplicamos el enfoque TDD, donde se deben diseñar y ejecutar las pruebas unitarias de bajo nivel (que también deben fallar al principio, y una vez que se haga la codificación necesaria para implementar la funcionalidad, se debe lograr que estas pruebas pasen, se refactoriza el código y se corrigen los Bugs según corresponda.
- En este punto, podemos ejecutar las pruebas funcionales definidas bajo el enfoque BDD, verificar que pasen y en caso contrario refactorizar. En un entorno CI/CD, cuando se integra el nuevo código o evolutivo, se ejecutan automáticamente las unitarias (bajo nivel), seguidamente las funcionales o E2E (alto nivel), y se realizan los ajustes o refactorizaciones correspondientes en cada nivel hasta lograr que todas se ejecuten exitosamente. Es un proceso iterativo y se debe repetir por cada historia de usuario dentro de un sprint.

Integrar TDD en BDD en el marco agile es una práctica que en principio no es fácil de adoptar, requiere crear el hábito, romper con el enfoque tradicional de hacer las cosas e ir optimizando

el proceso sobre la marcha. Es un enfoque iterativo que se alinea con las metodologías ágiles como Scrum y la filosofía Lean, sin duda puede mejorar la fluidez del desarrollo y las pruebas, permitiendo obtener un “feedback” temprano, detectar y corregir los defectos para asegurar la calidad del desarrollo.

7.6 Combinación de pruebas manuales y automatizadas

El plan de pruebas debe incluir tanto pruebas automáticas como manuales. Es fundamental determinar el tipo de prueba adecuado para cada aspecto y etapa del producto.

Las pruebas manuales abarcan una amplia variedad de condiciones y escenarios, proporcionando valiosas observaciones sobre la experiencia del usuario. Es ideal contemplar pruebas exploratorias, de usabilidad y ad hoc, realizadas por testers con diferentes perfiles de usuarios y dispositivos.

Se debe considerar optimizar el tiempo, por lo que recomendamos complementar con pruebas automatizadas, especialmente aquellas que formen parte del flujo CI&CD, por ejemplo: Pruebas Unitarias o de integración entre componentes, así como también, las pruebas no funcionales que tengan lugar (seguridad, estrés, carga, rendimiento, etc.)

7.7 Comunicación y trabajo en equipo

Esta práctica se extiende a todo el equipo, se centra en promover la comunicación regular y el esfuerzo colectivo en las actividades de prueba entre los miembros del equipo, es fundamental evolucionar la cultura QA y lograr una comprensión integral de la estrategia de pruebas. Como resultado, disminuyen los defectos, riesgos y vulnerabilidades, aumentando la calidad del desarrollo.

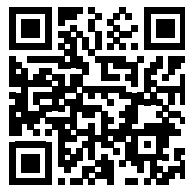
7.8 Reporte y Monitoreo de Resultados

Es recomendable implementar una estrategia para el reporte de resultados de las pruebas de manera eficaz (en lo posible de forma automatizada), debe incluir detalles relevantes sobre la cobertura de pruebas, el estado de las pruebas, los defectos encontrados, y la calidad general del software. Este enfoque facilita la toma de decisiones oportunamente y permite realizar un seguimiento adecuado de las actividades de prueba en tiempo real.

**Para cualquier duda o sugerencia,
puedes contactar con el equipo
de QA de Izertis**



Eneritz Zubizarreta Saenz de Zaitegui
Director of Quality Assurance



izertis