Testing I

**Certified Tech Developer**

**PRIMER AÑO**

**SEGUNDO BIMESTRE**

Tabla de contenido

[Módulo 1: Fundamentos del Testing 6](#_Toc78190496)

[Introducción al Testing 6](#_Toc78190497)

[Principios del Testing 6](#_Toc78190498)

[Tester 7](#_Toc78190499)

[Aspecto psicológico del testing 7](#_Toc78190500)

[Prueba independiente 7](#_Toc78190501)

[Mesa de 3 patas 8](#_Toc78190502)

[Ciclo de vida de las pruebas de software 8](#_Toc78190503)

[Pruebas 11](#_Toc78190504)

[Tipos de Enfoques 11](#_Toc78190505)

[Niveles de prueba 12](#_Toc78190506)

[Tipos de Prueba 16](#_Toc78190507)

[Módulo 2: Gestión de Defectos 17](#_Toc78190508)

[Defectos 17](#_Toc78190509)

[¿Qué es un defecto? 17](#_Toc78190510)

[Ciclo de vida de un defecto 17](#_Toc78190511)

[Gestión de defectos 18](#_Toc78190512)

[Escribir un informe de defectos 18](#_Toc78190513)

[Módulo 3: Diseño de prueba 19](#_Toc78190514)

[Casos de prueba 19](#_Toc78190515)

[Características de un buen caso de prueba 19](#_Toc78190516)

[¿Qué debe contener un caso de prueba? 19](#_Toc78190517)

[Testing Negativo y Positivo 20](#_Toc78190518)

[Happy Path 20](#_Toc78190519)

[Caso de uso vs. Caso de prueba 20](#_Toc78190520)

[¿Qué es un caso de uso? 20](#_Toc78190521)

[¿Qué es un caso de prueba? 20](#_Toc78190522)

[¿Cómo combinamos casos de uso con casos de prueba? 20](#_Toc78190523)

[¿Qué son las pruebas de casos de uso? 21](#_Toc78190524)

[Técnicas de Prueba 21](#_Toc78190525)

[Clasificación de las técnicas de prueba 21](#_Toc78190526)

[Técnicas de caja negra 22](#_Toc78190527)

[Técnicas basadas en la experiencia 23](#_Toc78190528)

[Prueba exploratoria 23](#_Toc78190529)

[Prueba basada en listas de comprobación 23](#_Toc78190530)

[Módulo 4: Implementación de la prueba 23](#_Toc78190531)

[Ejecución de los casos de prueba 23](#_Toc78190532)

[Creación de suites 24](#_Toc78190533)

[Módulo 5: Análisis de la Prueba 24](#_Toc78190534)

[Pruebas Estáticas y Dinámicas 24](#_Toc78190535)

[Pruebas Estáticas 24](#_Toc78190536)

[Pruebas Dinámicas 25](#_Toc78190537)

[Proceso de Revisión 26](#_Toc78190538)

[Roles 26](#_Toc78190539)

[Tipos 26](#_Toc78190540)

[Técnicas 27](#_Toc78190541)

[Actividades 27](#_Toc78190542)

[Requerimientos 27](#_Toc78190543)

[Requisitos funcionales 27](#_Toc78190544)

[Requisitos no funcionales 27](#_Toc78190545)

[Módulo 6: Planificación de la prueba 28](#_Toc78190546)

[Entornos de prueba 28](#_Toc78190547)

[DEV 28](#_Toc78190548)

[QA 28](#_Toc78190549)

[UAT 29](#_Toc78190550)

[STG 30](#_Toc78190551)

[Métricas y reportes 30](#_Toc78190552)

[Métrica del producto 31](#_Toc78190553)

[Métricas de proceso 32](#_Toc78190554)

[Módulo 7: Introducción a TDD 33](#_Toc78190555)

[Debug 33](#_Toc78190556)

[Breakpoints 33](#_Toc78190557)

[Interfaz 34](#_Toc78190558)

[Debug desde la consola de Chrome 34](#_Toc78190559)

[Debugging vs Testing 34](#_Toc78190560)

[Prueba de componente o prueba unitaria (unit test) 35](#_Toc78190561)

[Prueba de componentes 36](#_Toc78190562)

[Generalidades 36](#_Toc78190563)

[Ventajas 37](#_Toc78190564)

[Frameworks 37](#_Toc78190565)

[Test unitario 3 con JavaScript 37](#_Toc78190566)

[Técnicas de prueba de caja blanca 38](#_Toc78190567)

[TDD - Test Driven Development o Desarrollo guiado por pruebas 39](#_Toc78190568)

[TDD vs BDD 40](#_Toc78190569)

[Mock y Stub 41](#_Toc78190570)

[Cobertura de pruebas con JEST 41](#_Toc78190571)

[Módulo 8: BE testing 43](#_Toc78190572)

[Introducción API Testing 43](#_Toc78190573)

[¿Qué es el testing de back end? 43](#_Toc78190574)

[Introducción a APIs 43](#_Toc78190575)

[Testing de APIs 43](#_Toc78190576)

[HTTP y sus métodos 43](#_Toc78190577)

[API Testing 44](#_Toc78190578)

[Método GET 44](#_Toc78190579)

[Método POST 46](#_Toc78190580)

[API’s Testing - Automatización de pruebas 48](#_Toc78190581)

[API Testing- Creación de pruebas (JS) 48](#_Toc78190582)

[Colecciones y variables de entorno con Postman 51](#_Toc78190583)

[Colecciones 51](#_Toc78190584)

[Variables de entorno 52](#_Toc78190585)

[Runner para correr colecciones 52](#_Toc78190586)

[Módulo 9: Introducción a la Automatización 52](#_Toc78190587)

[Pruebas automatizadas 52](#_Toc78190588)

[Tipos de pruebas automatizadas 53](#_Toc78190589)

[Objetivos de automatización 53](#_Toc78190590)

[Ventajas y Desventajas 53](#_Toc78190591)

[Patrones de Diseño 53](#_Toc78190592)

[Patrones de diseño en automatización 53](#_Toc78190593)

[¿Qué son los principios SOLID? 54](#_Toc78190594)

[Page Object Model 55](#_Toc78190595)

[Introducción a Selenium 56](#_Toc78190596)

[Selenium IDE 56](#_Toc78190597)

[Selenium WebDriver 57](#_Toc78190598)

[Instalación de Selenium WebDriver 58](#_Toc78190599)

[Creación del primer script de prueba: Login de Facebook 58](#_Toc78190600)

[Módulo 10: Full stack y DevOps tester 59](#_Toc78190601)

[Full Stack Tester 59](#_Toc78190602)

[Frameworks 60](#_Toc78190603)

[Tipos de Frameworks 60](#_Toc78190604)

[Beneficios Generales 62](#_Toc78190605)

[Test Plan 63](#_Toc78190606)

[Plan de Pruebas 63](#_Toc78190607)

[5 pasos para crear un Test Plan 64](#_Toc78190608)

[Template: Plan de Testing 65](#_Toc78190609)

[¿Qué es DevOps? 68](#_Toc78190610)

[Continuous Integration 69](#_Toc78190611)

[Continuous delivery 69](#_Toc78190612)

[Continuous Deployment 69](#_Toc78190613)

[Relación entre las prácticas CI/CD 70](#_Toc78190614)

[Herramientas 70](#_Toc78190615)

[¿Quiénes participan en las pruebas de DevOps? 71](#_Toc78190616)

Clase 1: Primeros Pasos

# Módulo 1: Fundamentos del Testing

## Introducción al Testing

Como testers probamos si funciona todo de acuerdo a lo esperado y, en caso contrario, avisamos para que se pueda solucionar.

Un tester parte siempre de la suposición de que el programa contiene errores por lo que deberá probarlo para encontrar la mayor cantidad y avisar para que lo solucionen. De esta manera el Testing brinda calidad a esos desarrollos. Pero, ¿a qué nos referimos cuando hablamos de calidad? La calidad es la satisfacción del cliente ante un producto o servicio.

Con la información que nos brinda el testing podemos inferir sobre el sistema que está bajo prueba, por ejemplo, saber cuándo está listo para ser liberado. Con estas pruebas sabremos cuánto podemos confiar en el software. Pensemos en esto, si una aplicación no te da seguridad porque puede interferir en tu privacidad ¿la descargarías?

**Realizar pruebas frecuentes es vital en el proceso de desarrollo**; sin embargo, hay factores externos que hacen que más allá de haber realizado las pruebas adecuadas, se estén llevando a cabo sobre requerimientos equivocados. Y aquí llegamos al factor humano, existen proyectos en la fase final que tienen que volver a corregirse por motivos humanos, uno de los más recurrentes es que el representante del cliente ante el proyecto no fue lo suficientemente claro con los requerimientos que el cliente haya decidido modificar, agregar requisitos o que, desde el liderazgo del proyecto, se hayan malentendido los mismos. También puede haber errores de diseño o que los testers no estén suficientemente capacitados.

Como conclusión el Testing sirve para encontrar el mayor número de defectos y remediarlos, asegurar que el producto funciona tal como se ha definido en los requisitos y proporcionar al producto final un mayor grado de calidad.

De lo visto anteriormente, se desprende que, si bien la ingeniería de software es una disciplina relativamente joven, durante su corta vida ha atravesado diferentes etapas de las cuales aprendió la lección: **la calidad es un valor en sí mismo y no un gasto que las empresas deben realizar para que su negocio prospere**.

Sin embargo, se dice que esta ingeniería “es como una persona de la cual todos alguna vez se enamoran, pero con la que que muy pocos están dispuestos a casarse” porque es muy común encontrarse con proyectos que generan productos de dudosa calidad debido a que los responsables decidieron no realizar tareas de revisión de diseño y código o determinadas pruebas porque dilatan el tiempo de salida al mercado, cuestan dinero y además no habrá diferencia entre uno y otro producto cuando el proyecto termine.

### Principios del Testing

La prueba muestra la presencia de defectos, no su ausencia: No puede probar que no hay defectos. Reduce la probabilidad de que queden defectos no descubiertos en el software, pero, incluso si no se encuentran, el proceso de prueba no es una demostración de corrección.

La prueba exhaustiva es imposible: No es posible probar todo —todas las combinaciones de entradas y precondiciones—, excepto en casos triviales. En lugar de intentar realizar pruebas exhaustivas se deberían utilizar el análisis de riesgos, las técnicas de prueba y las prioridades para centrar los esfuerzos de prueba.

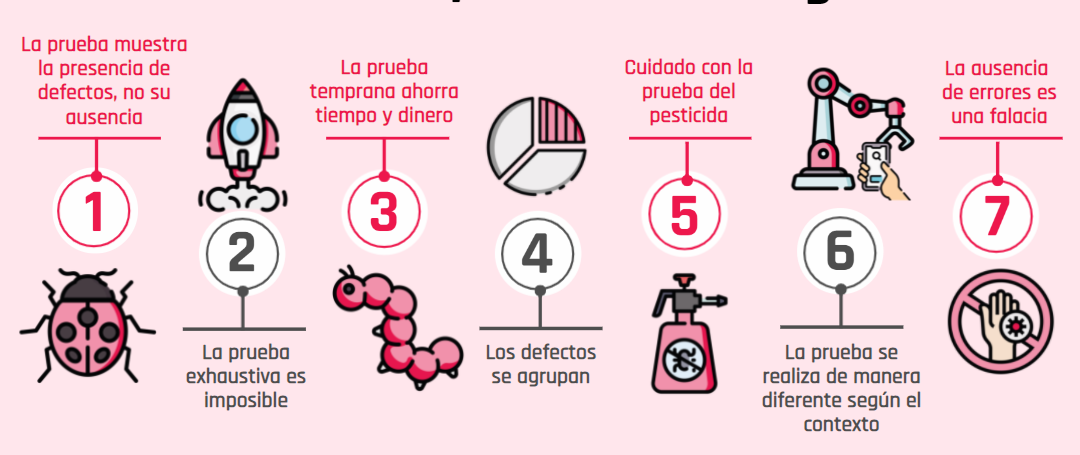
La prueba temprana ahorra tiempo y dinero: Para detectar defectos de forma temprana, las actividades de testing, tanto estáticas como dinámicas, deben iniciarse lo antes posible en el ciclo de vida de desarrollo de software para ayudar a reducir o eliminar cambios costosos.

Los defectos se agrupan: En general, un pequeño número de módulos contiene la mayoría de los defectos descubiertos durante la prueba previa al lanzamiento o es responsable de la mayoría de los fallos operativos.

Cuidado con la prueba del pesticida: Si las mismas pruebas se repiten una y otra vez, eventualmente estas pruebas ya no encontrarán ningún defecto nuevo. Para detectarlo, es posible que sea necesario cambiar las pruebas y los datos de prueba existentes.

La prueba se realiza de manera diferente según el contexto: Por ejemplo, el software de control industrial de seguridad crítica se prueba de forma diferente a una aplicación móvil de comercio electrónico.

La ausencia de errores es una falacia: El éxito de un sistema no solo depende de encontrar errores y corregirlos hasta que desaparezcan ya que puede no haber errores, pero sí otros problemas. Existen otras variables a tener en cuenta al momento de medir el éxito.



## Tester

El testing es el proceso de ejecución de un programa con la intención de encontrar errores.

### Aspecto psicológico del testing

Los seres humanos tienden a ser sumamente orientados a objetivos y el establecimiento de la meta adecuada tiene un efecto psicológico importante. Si nuestro objetivo es demostrar que un programa no tiene errores, entonces, subconscientemente estaremos dirigidos a esa meta, es decir, tendemos a seleccionar los datos de prueba que tienen una baja probabilidad de causar que el programa falle.

Por otro lado, si nuestro objetivo es demostrar que un programa tiene errores, nuestros datos de prueba tendrán una mayor probabilidad de encontrarlos.

Más allá del desarrollador o el tester, las tareas de prueba pueden ser realizadas por personas que desempeñan un rol de prueba específico u otro rol —por ejemplo, clientes—.

### Prueba independiente

La forma en que se implementa la independencia de la prueba varía dependiendo del modelo de ciclo de vida de desarrollo de software.

Por ejemplo, en el desarrollo ágil, los probadores pueden formar parte de un equipo de desarrollo. En algunas organizaciones que utilizan métodos ágiles, estos probadores también pueden ser considerados parte de un equipo de prueba independiente más grande. Además, en dichas organizaciones, los propietarios de producto pueden realizar la prueba de aceptación para validar las historias de usuario al final de cada iteración.

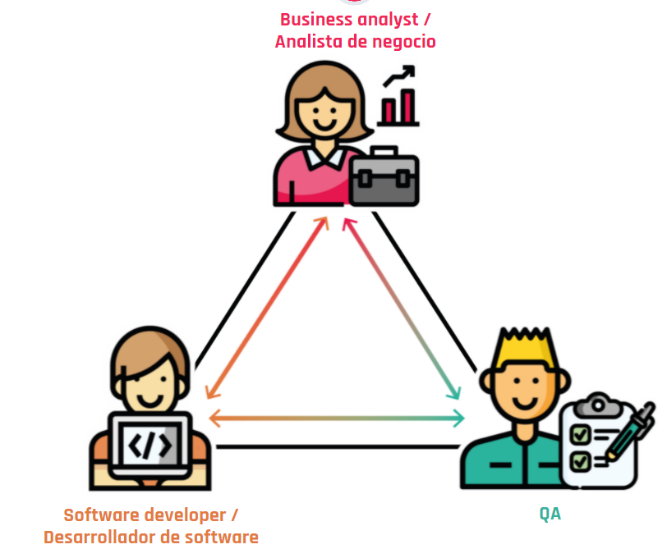
|  |  |
| --- | --- |
| Ventajas | Desventajas |
| Es probable que los probadores independientes reconozcan diferentes tipos de fallos en comparación con los desarrolladores debido a sus diferentes contextos, perspectivas técnicas y sesgos. | Los desarrolladores pueden perder el sentido de la responsabilidad con respecto a la calidad. |
| Un probador independiente puede verificar, cuestionar o refutar las suposiciones hechas por los implicados durante la especificación e implementación del sistema. | Los probadores independientes pueden ser vistos como un cuello de botella o ser culpados por los retrasos en el lanzamiento o liberación. |
|  | Los probadores independientes pueden carecer de información importante —por ejemplo, sobre el objeto de prueba—. |

Tabla 1: Ventajas y Desventajas de la independencia de pruebas

### Mesa de 3 patas

Si bien cada actor tiene un rol definido, es necesario un trabajo en comunión entre los 3 actores. Es decir, es necesario que trabajen como equipo. Por eso, utilizamos la analogía con una mesa de 3 patas, pues si falta alguna de ellas, la mesa no podría estar de pie.

En algunas empresas de software pequeñas o “start up” es posible que una misma persona tenga más de un rol.

Business analyst / Analista de negocio: Se encarga de detectar los factores clave del negocio y es el intermediario entre el departamento de sistemas y el cliente final.

QA: La principal función es probar los sistemas informáticos para que funcionen correctamente de acuerdo a los requerimientos del cliente, documentar los errores encontrados y desarrollar procedimientos de prueba para hacer un seguimiento de los problemas de los productos de forma más eficaz y eficiente.

Software developer / Desarrollador de software: Su función es diseñar, producir, programar o mantener componentes o subconjuntos de software conforme a especificaciones funcionales y técnicas para ser integrados en aplicaciones.

### Ciclo de vida de las pruebas de software

No existe un proceso de prueba único y universal, pero existen actividades de prueba comunes que nos ayudan a organizarnos para alcanzar los objetivos establecidos.

#### Ciclo de Deming o Ciclo PHVA

Es la metodología más usada para solucionar problemas y ejecutar sistemas de mejora continua. Su aplicación ayuda a que las organizaciones mejoren su rendimiento y aumenten su productividad.

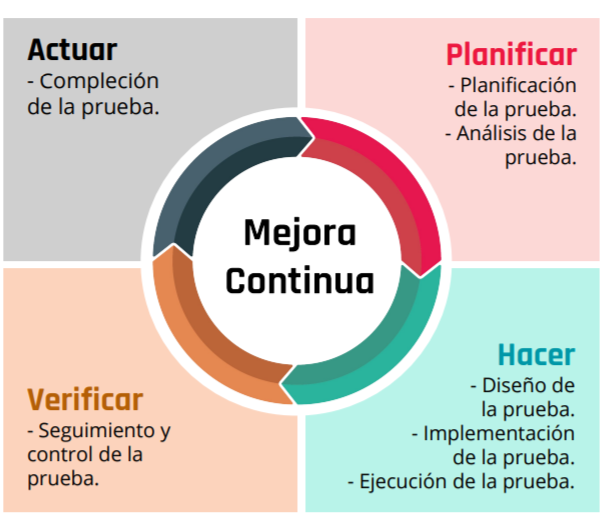
Se compone cuatro etapas cíclicas de forma que una vez acabada la etapa final se debe volver a la primera y repetir el ciclo de nuevo. Estas etapas son: planificar, hacer, verificar y actuar.

#### Ciclo de vida de las pruebas de software (STLC)

Algunos factores de contexto que influyen en el proceso de prueba son:

* Modelo de ciclo de vida de desarrollo de software y metodologías de proyecto en uso.
* Niveles y tipos de prueba considerados.
* Riesgos de producto y de proyecto.
* Dominio del negocio.
* Restricciones operativas, incluyendo, pero no limitadas a: Plazos y complejidad.

Conociendo la importancia y eficiencia que tiene la aplicación del Ciclo de Deming y teniendo en cuenta que persigue el mismo objetivo que el ciclo de vida de la prueba, es decir, “La entrega de un producto de calidad, a través de la mejora continua de sus procesos”, se pueden ubicar las actividades del ciclo de vida de prueba de software en el ciclo de Deming de la siguiente forma:



El ciclo de vida de las pruebas de software consiste en las siguientes actividades principales (aunque no siempre están agrupadas de esta manera en todos los proyectos de software):

1. Planificación
2. Seguimiento y control
3. Análisis
4. Diseño
5. Implementación
6. Ejecución
7. Conclusión

##### Planificación

En esta actividad se definen los objetivos y el enfoque de la prueba dentro de las restricciones impuestas por el contexto.

Algunas subactividades realizadas son:

* Determinar el alcance, los objetivos y los riesgos.
* Definir el enfoque y estrategia general.
* Integrar y coordinar las actividades a realizar durante el ciclo de vida del software.
* Definir las especificaciones de técnicas, tareas de prueba adecuadas, las personas y otros recursos necesarios.
* Establecer un calendario de pruebas para cumplir con un plazo límite.
* Generar el plan de prueba.

Documentos de salida: Plan de prueba —general y/o por nivel de prueba—.

##### Seguimiento y control

El objetivo de esta actividad es reunir información y proporcionar retroalimentación y visibilidad sobre las actividades de prueba. Como parte del control, se pueden tomar acciones correctivas, como cambiar la prioridad de las pruebas, el calendario y reevaluar los criterios de entrada y salida.

Algunas subactividades realizadas son:

* Comprobar los resultados y los registros de la prueba en relación con los criterios de cobertura especificados.
* Determinar si se necesitan más pruebas dependiendo del nivel de cobertura que se debe alcanzar.

Documento de salida: Informe de avance de la prueba

##### Análisis

Durante esta actividad se determina “qué probar”.

Algunas subactividades realizadas son:

* Analizar la base de prueba correspondiente al nivel de prueba considerado —información de diseño e implementación, la implementación del componente o sistema en sí, Informes de análisis de riesgos, etc.—.
* Identificar defectos de distintos tipos en las bases de prueba —ambigüedades, omisiones, inconsistencias, inexactitudes, etc—.
* Identificar los requisitos que se van a probar y definir las condiciones de prueba para cada requisito.
* Captura de la trazabilidad entre la base de prueba y las condiciones de prueba.

Documento de salida: Contratos de prueba que contienen las condiciones de prueba

##### Diseño

Durante esta actividad se determina “cómo probar”.

Algunas subactividades realizadas son:

* Diseñar y priorizar casos de prueba y conjuntos de casos de prueba de alto nivel.
* Identificar los datos de prueba necesarios.
* Diseñar el entorno de prueba e identificar la infraestructura y las herramientas necesarias.
* Capturar la trazabilidad base de prueba, las condiciones de prueba, los casos de prueba y los procedimientos de prueba

Documento de salida: Casos de prueba de alto nivel diseñados y priorizados.

##### Implementación

Se completan los productos de prueba necesarios para la ejecución de la prueba, incluyendo la secuenciación de los casos de prueba en procedimientos de prueba.

Algunas subactividades realizadas son:

* Desarrollar y priorizar procedimientos de prueba.
* Crear juegos de prueba (test suite) a partir de los procedimientos de prueba.
* Organizar los juegos de prueba dentro de un calendario de ejecución.
* Construir el entorno de prueba y verificar que se haya configurado correctamente todo lo necesario.
* Preparar los datos de prueba y asegurarse de que estén correctamente cargados.
* Verificar y actualizar la trazabilidad entre la base de prueba, las condiciones de prueba, los casos de prueba, los procedimientos de prueba y los juegos de prueba.

Documento de Salida: Procedimientos y datos de prueba, calendario de ejecución, test suite.

##### Ejecución

Durante esta actividad se realiza la ejecución de los casos de prueba.

Algunas subactividades realizadas son:

* Registrar los identificadores y las versiones de los elementos u objetos de prueba.
* Ejecutar y registrar el resultado pruebas de forma manual o utilizando herramientas.
* Comparar los resultados reales con los resultados esperados.
* Informar sobre los defectos en función de los fallos observados.
* Repetir las actividades de prueba, ya sea como resultado de una acción tomada para una anomalía o como parte de la prueba planificada —retest o prueba de confirmación—.
* Verificar y actualizar la trazabilidad entre la base de prueba, las condiciones de prueba, los casos de prueba, los procedimientos de prueba y los resultados de la prueba.

Documento de salida: Reporte de defectos, informe de ejecución de pruebas.

##### Conclusión

Se recopilan la información de las actividades completadas y los productos de prueba. Puede ocurrir cuando un sistema software es liberado, un proyecto de prueba es completado —o cancelado—, finaliza una iteración de un proyecto ágil, se completa un nivel de prueba o se completa la liberación de un mantenimiento.

Algunas subactividades realizadas son:

* Comprobar que todos los informes de defecto están cerrados.
* Finalizar, archivar y almacenar el entorno de prueba, los datos de prueba, la infraestructura de prueba y otros productos de prueba para su posterior reutilización.
* Traspaso de los productos de prueba a otros equipos que podrían beneficiarse con su uso.
* Analizar las lecciones aprendidas de las actividades de prueba completadas.
* Utilizar la información recopilada para mejorar la madurez del proceso de prueba.

Documento de salida: Informe resumen de prueba, lecciones aprendidas.

Clase 2: Fundamentos del Testing

## Pruebas

### Tipos de Enfoques

|  |  |
| --- | --- |
| ENFOQUE TRADICIONAL | ENFOQUE ÁGIL |
| Etapas de procesos de larga duración y secuenciales | Iteraciones cortas definidas como sprints |
| Pruebas estáticas al principio sobre los documentos que son bases de prueba. Las pruebas dinámicas se realizan como una actividad al final, luego de tener la solución codificada | Pruebas dinámicas y continuas durante la iteración. Los requerimientos son analizados como prestaciones (features) y se crean escenarios de prueba usando un enfoque de desarrollo guiado por comportamiento (BDD) |
| Se realiza una prueba de ambigüedad sobre los requerimientos | Se dividen las prestaciones en historias de usuarios (user stories) y se modelan escenarios de prueba para esas historias |
| Se dividen los requerimientos en historias pequeñas | Se escriben los unit test y luego se codifica la solución usando un enfoque de desarrollo guiado por pruebas (TDD) |
| Se desarrolla el código en base a la documentación de diseño. Luego de tener el código se desarrollan los unit test | Se enfoca en la ligera documentación priorizando lo que entregue valor al cliente |
| Existe mayor documentación de cada proceso | Se ejecutan pruebas continuamente y desde etapas tempranas. Toma relevancia la aplicación de integración continua (CI) y distribución o desarrollo continuo (CD) |
| Las pruebas se ejecutan luego de tener el código generado de todo un requerimiento o todo el sistema | Adquieren gran protagonismo las pruebas automatizadas |
| Las pruebas son generalmente manuales | Además, la integración continua permite tener una retroalimentación activa. |

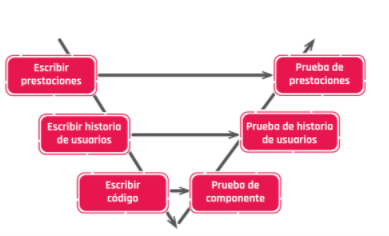


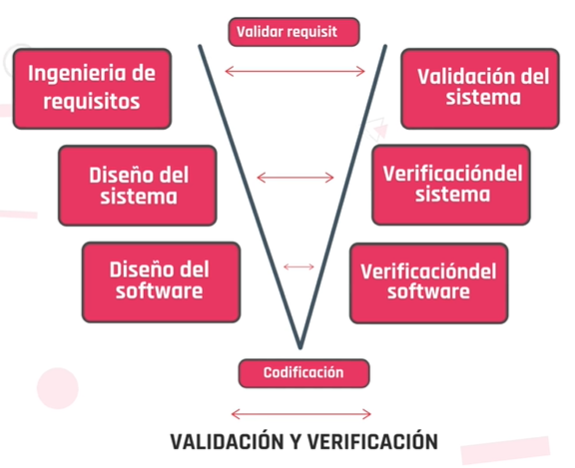
Ilustración 1: Enfoque Tradicional



Ilustración 2: Enfoque Ágil

### Niveles de prueba

Cuando se desarrolla una aplicación de software hay que llevar a cabo pruebas en todos los niveles para entregar al cliente un producto que cumpla sus expectativas y necesidades, asegurando la calidad del mismo. Para entregar un producto de calidad y confiable debemos testearlo, para esto existen distintos niveles de prueba o actividades que se organizan y gestionan conjuntamente.

Para comprender que son niveles de prueba vamos a utilizar el modelo en V que es empleado en diversos procesos de desarrollo. En los años 90 aparece su primera versión, pero con el tiempo se ha ido perfeccionando y adaptando a los métodos modernos de desarrollo. Además de las fases de desarrollo de un proyecto, el modelo V también define los procedimientos de gestión de la calidad que lo acompañan y describe cómo pueden interactuar estas fases individuales entre sí. Este modelo compara las fases de desarrollo con las fases de control de la calidad correspondientes.

Del lado izquierdo podemos observar las fases de desarrollo y diseño y, para cada una de ellas, del lado derecho se encuentran las medidas de control de calidad. En la unión de ambos lados se sitúa la implementación del producto.

Comenzando del lado izquierdo, cuando empieza el proyecto, el modelo prevé un análisis de requisitos y especificaciones. Luego el proyecto se centra en las características funcionales y no funcionales, posteriormente se definen los componentes y las interfaces y finalmente se realiza la implementación.

De acuerdo estas etapas, el desarrollo del software se produce a través de diferentes niveles de control de la calidad o también llamadas de verificación o validación que siempre están relacionadas con cada una de las fases de desarrollo.

*Verificación: ¿Estamos construyendo el producto correctamente?*

*Validación: ¿Estamos construyendo el producto correcto?*

Verificación es el conjunto de actividades que aseguran que el software implemente correctamente una función específica y, validación, es un conjunto diferente de actividades que aseguran que el software construido respeta los requisitos del cliente.

El método V abarca las pruebas de componentes, de integración, de sistema y de aceptación.

En la **prueba de componente** se aísla cada parte del programa: clases, módulos, objetos, paquetes, subsistemas para comprobar que cada una de esas partes funciona correctamente por separado. Por otro lado, se hacen las **pruebas de integración** donde se realizan sobre un conjunto de módulos de la aplicación para comprobar que funcionen correctamente, es decir, se prueba la interacción entre las distintas partes del Software. Luego las **pruebas de sistemas** son llevadas a cabo por el proveedor en un entorno de laboratorio para demostrar que el sistema cumple con los requisitos funcionales y no funcionales y con el diseño técnico claro; están diseñadas para probar el sistema de su totalidad. Por último, las **pruebas de aceptación** son realizadas por el usuario en un entorno muy similar al de producción, para demostrar que el sistema cumple las especificaciones funcionales y requisitos del cliente.

#### Prueba de Componente

Objetivos Específicos

* Reducir el riesgo.
* Verificar que los comportamientos funcionales y no funcionales del componente son los diseñados y especificados.
* Generar confianza en la calidad del componente.
* Encontrar defectos en el componente.
* Prevenir la propagación de defectos a niveles

Base de prueba

Algunos ejemplos de productos de trabajo que se pueden utilizar como base de prueba incluyen: Diseño detallado, código, modelo de datos, especificaciones de los componentes.

Objetos de prueba

Los objetos de prueba característicos para la prueba de componente incluyen: Componentes, unidades o módulos. Código y estructuras de datos, clases, módulos de base de datos.

Defectos y fallos característicos

Ejemplos de defectos y fallos característicos de la prueba de componente incluyen: Funcionamiento incorrecto (por ejemplo, no lo hace de la manera en que se describe en las especificaciones de diseño).

Problemas de flujo de datos, código y lógica incorrectos.

Enfoque y responsabilidades específicas

En general, el desarrollador que escribió el código realiza la prueba de componente. Los desarrolladores pueden alternar el desarrollo de componentes con la búsqueda y corrección de defectos.

A menudo, estos escriben y ejecutan pruebas después de haber escrito el código de un componente. Sin embargo, especialmente en el desarrollo ágil, la redacción de casos de prueba de componente automatizados puede preceder a la redacción del código de la aplicación.

#### Prueba de Integración

Objetivos

La prueba de integración se centra en las interacciones entre componentes o sistemas.

* Reducir el riesgo.
* Verificar que los comportamientos funcionales y no funcionales de las interfaces sean los diseñados y especificados.
* Generar confianza en la calidad de las interfaces.
* Encontrar defectos —que pueden estar en las propias interfaces o dentro de los componentes o sistemas—.
* Prevenir la propagación de defectos a niveles de prueba superiores.

Base de prueba

Algunos ejemplos de productos de trabajo que pueden utilizarse como base de prueba para la prueba de integración incluyen: Diseño de software y sistemas, diagramas de secuencia, especificaciones de interfaz y protocolos de comunicación, casos de uso, arquitectura a nivel de componente o de sistema, flujos de trabajo, definiciones de interfaces externas.

Objetos de prueba

Los objetos de prueba característicos para la prueba de integración incluyen: Subsistemas, bases de datos, infraestructura, interfaces, interfaces de programación de aplicaciones (API por sus siglas en inglés), microservicios.

Defectos y fallos característicos

* Datos incorrectos, datos faltantes o codificación incorrecta de datos.
* Secuenciación o sincronización incorrecta de las llamadas a la interfaz.
* Incompatibilidad de la interfaz.
* Fallos en la comunicación entre componentes.
* Fallos de comunicación entre componentes no tratados o tratados de forma incorrecta.
* Suposiciones incorrectas sobre el significado, las unidades o las fronteras de los datos que se transmiten entre componentes.

Enfoque y responsabilidades específicas

La prueba de integración debe concentrarse en la integración propiamente dicha. Se puede utilizar los tipos de prueba funcional, no funcional y estructural. En general es responsabilidad de los testers.

#### Prueba de Sistema

Objetivos

* Reducir el riesgo.
* Verificar que los comportamientos funcionales y no funcionales del sistema son los diseñados y especificados.
* Validar que el sistema está completo y que funcionará como se espera.
* Generar confianza en la calidad del sistema considerado como un todo.
* Encontrar defectos.
* Prevenir la propagación de defectos a niveles de prueba superiores o a producción.

Base de prueba

Algunos ejemplos de productos de trabajo que se pueden utilizar como base de prueba incluyen: Especificaciones de requisitos del sistema y del software (funcionales y no funcionales), informes de análisis de riesgo, casos de uso, épicas e historias de usuario, modelos de comportamiento del sistema, diagramas de estado, manuales del sistema y del usuario.

Objetos de prueba

* Aplicaciones.
* Sistemas hardware/software.
* Sistemas operativos.
* Sistema sujeto a prueba (SSP).
* Configuración del sistema y datos de configuración.

Defectos y fallos característicos

* Cálculos incorrectos.
* Comportamiento funcional o no funcional del sistema incorrecto o inesperado.
* Control y/o flujos de datos incorrectos dentro del sistema.
* Incapacidad para llevar a cabo, de forma adecuada y completa, las tareas funcionales extremo a extremo.
* Fallo del sistema para operar correctamente en el/los entorno/s de producción.
* Fallo del sistema para funcionar como se describe en los manuales del sistema y de usuario.

Enfoque y responsabilidades específicas

La prueba de sistema debe centrarse en el comportamiento global y extremo a extremo del sistema en su conjunto, tanto funcional como no funcional. Debe utilizar las técnicas más apropiadas para los aspectos del sistema que serán probados. Los probadores independientes, en general, llevan a cabo la prueba de sistema

#### Prueba de Aceptación

Objetivos

La prueba de aceptación, al igual que la prueba de sistema, se centra normalmente en el comportamiento y las capacidades de todo un sistema o producto. Los objetivos de la prueba de aceptación incluyen: Establecer confianza en la calidad del sistema en su conjunto, validar que el sistema está completo y que funcionará como se espera. Verificar que los comportamientos funcionales y no funcionales del sistema sean los especificados.

Base de prueba

Entre los ejemplos de productos de trabajo que se pueden utilizar como base de prueba se encuentran:

* Procesos de negocio.
* Requisitos de usuario o de negocio.
* Normativas, contratos legales y estándares.
* Casos de uso.
* Requisitos de sistema.
* Documentación del sistema o del usuario.
* Procedimientos de instalación.
* Informes de análisis de riesgo.

Objetos de prueba

* Sistema sujeto a prueba.
* Configuración del sistema y datos de configuración.
* Procesos de negocio para un sistema totalmente integrado.
* Sistemas de recuperación y sitios críticos (para pruebas de continuidad del negocio y recuperación de desastres):
* Procesos operativos y de mantenimiento.
* Formularios.
* Informes.
* Datos de producción existentes y transformados.

Defectos y fallos característicos

Entre los ejemplos de defectos característicos de cualquier forma de prueba de aceptación se encuentran:

* Los flujos de trabajo del sistema no cumplen con los requisitos de negocio o de usuario.
* Las reglas de negocio no se implementan de forma correcta.
* El sistema no satisface los requisitos contractuales o reglamentarios.
* Fallos no funcionales tales como vulnerabilidades de seguridad, eficiencia de rendimiento inadecuada bajo cargas elevadas o funcionamiento inadecuado en una plataforma soportada.

Enfoque y responsabilidades específicas

A menudo es responsabilidad de los clientes, usuarios de negocio, propietarios de producto u operadores de un sistema, y otros implicados también pueden estar involucrados. La prueba de aceptación se considera, a menudo, como el último nivel de prueba en un ciclo de vida de desarrollo secuencial.

### Tipos de Prueba

Un tipo de prueba es un grupo de actividades de pruebas destinadas a probar las características específicas de un sistema de software, o de una parte de un sistema, basados en objetivos de pruebas específicas.

Dichos objetivos pueden incluir:

* Evaluar las características de calidad funcional tales como la completitud, corrección y pertinencia.
* Evaluar características no funcionales de calidad, tales como la fiabilidad, eficiencia de desempeño, seguridad, confiabilidad y usabilidad.

|  |  |
| --- | --- |
| Prueba Funcional | Prueba No Funcional |
| La prueba funcional de un sistema incluye pruebas que evalúan las funciones que el sistema debe realizar. Las funciones describen qué hace el sistema. | La prueba no funcional prueba “cómo de bien” se comporta el sistema. |
| La prueba funcional observa el comportamiento del software. | El diseño y ejecución de la prueba funcional puede implicar competencias y conocimientos especiales, como el conocimiento de las debilidades inherentes a un diseño o tecnología |

# Módulo 2: Gestión de Defectos

Clase 4: Gestión de Defectos

## Defectos

### ¿Qué es un defecto?

¿Es lo mismo decir error, defecto o falla? Muchas veces usamos estos términos queriendo expresar el mismo significado, pero en realidad no son lo mismo.

Los tres conceptos están relacionados porque uno depende del otro, un error se produce por la equivocación de una persona, esto causa un defecto en el software y provoca un fallo cuando la prueba se ejecuta. Por ejemplo, el desarrollador al escribir el código lo hace incorrectamente cometiendo un error, esto se convierte en algo incorrecto en el código, un defecto, que al ejecutarse provoca un mal funcionamiento, un fallo.



### Ciclo de vida de un defecto

1. **Nuevo/Inicial:** Se recopila la información y se registra el defecto.
2. **Asignado**: Si es un defecto válido y debe solucionarse se asigna al equipo de desarrollo, sino se puede rechazar o diferir (bug triage).

**Duplicado:** Si el defecto se repite o existe otro con una misma causa raíz.

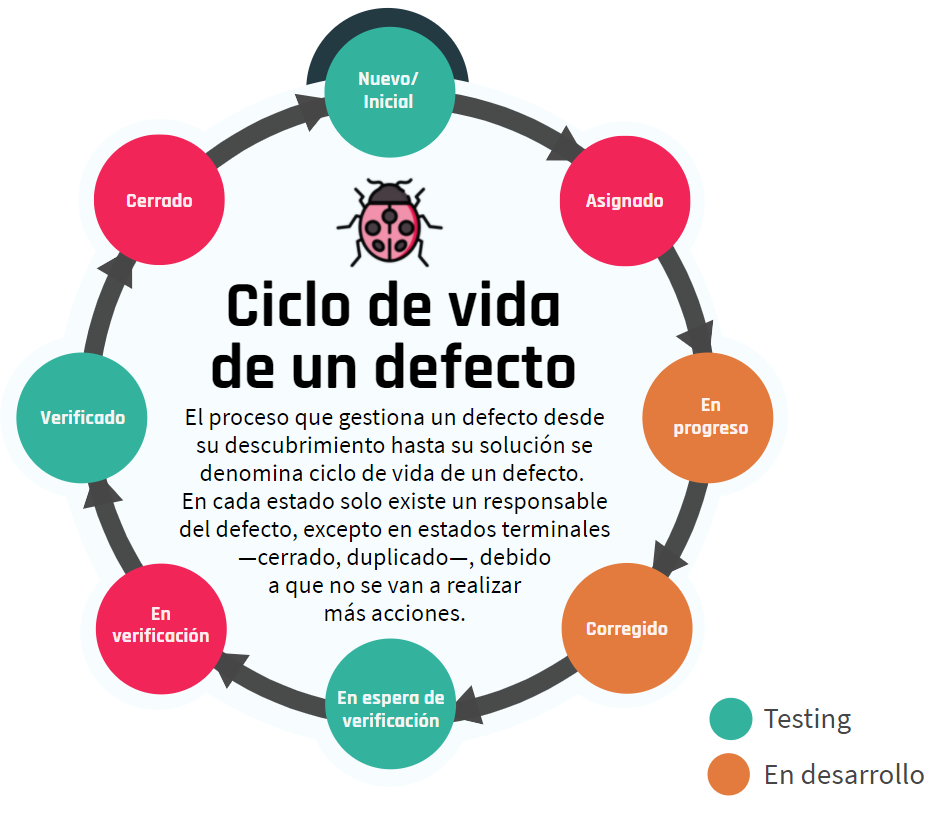
**Devuelto o rechazado:** Se solicita más información o el receptor rechaza el defecto.

**Diferido**: el defecto no es prioritario y se solucionará en una próxima versión.

1. **En progreso**: Se analiza y trabaja en la solución.
2. **Corregido:** Se realizan los cambios de código para solucionar el defecto.
3. **En espera de verificación**: En espera de que sea asignado a un probador. El desarrollador está a la expectativa del resultado de la verificación.
4. **En verificación:** El probador ejecuta una prueba de confirmación.

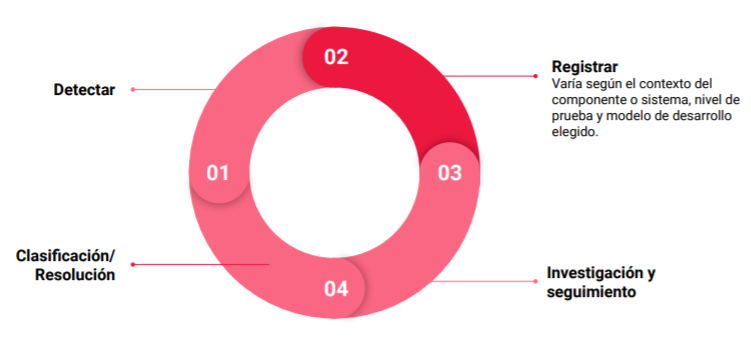
**Reabierto**: debe contener la siguiente descripción “La prueba de confirmación indica que el defecto no se ha solucionado.”

1. **Verificado:** Se obtiene el resultado esperado en la prueba de confirmación.
2. **Cerrado**: El defecto fue corregido y se encuentra disponible para el usuario final.



### Gestión de defectos

#### Proceso general



#### Objetivos

* Brindar información sobre cualquier evento adverso que haya ocurrido, para poder identificar efectos específicos, aislar el problema con una prueba de reproducción mínima y corregir los defectos potenciales.
* Proporcionar a los jefes de prueba un medio para hacer un seguimiento de la calidad del producto de trabajo y del impacto en la prueba.
* Dar ideas para la mejora de los procesos de desarrollo y prueba.

### Escribir un informe de defectos

Si el defecto se reporta eficientemente, las probabilidades de que sea solucionado rápidamente son mayores. Entonces, la solución de un defecto dependerá de la eficiencia con que se reporte.

¿Qué condiciones debemos tener en cuenta?

Los bugs deben tener identificadores únicos: Si bien muchas herramientas de bug tracking asignan automáticamente un ID único a los bugs, muchas veces se reportan fallas por medio de e-mails, saltando la registración en la herramienta.

Una falla debe ser reproducible para reportarla: Si el defecto no es reproducible, no es un defecto. Para defectos que ocurren en forma aislada, podemos realizarnos una nota personal para investigar luego y determinar qué condiciones deben ser dadas para que el mismo se produzca.

Ser específico: No se debe escribir suposiciones o ideas sobre lo que está ocurriendo u otra cosa que no sea la información relevante para poder reproducir el defecto.

Reportar cada paso realizado para reproducirlo: Toda la información que podamos darle al desarrollador para que pueda reproducir la falla siempre será bienvenida, no debemos obviar ningún paso que sea relevante para llegar al error en cuestión.

¿Cuáles son los problemas más comunes con los informes de defectos?

* Redactar un defecto de manera excesivamente coloquial y ambigua.
* Dar solo una captura del defecto sin indicar qué se estaba haciendo cuando sucedió.
* No incluir en la descripción del defecto cuál era el resultado esperado para los pasos realizados.
* No determinar un patrón con el cual el defecto ocurre antes de reportar el mismo (es importante para ser directos en cuál es el problema).
* No leer el defecto reportado siguiendo los pasos uno mismo para ver que la descripción es clara.
* No incluir información que, dada las características del defecto, la misma es de relevancia.

# Módulo 3: Diseño de prueba

Clase 5: Diseño de la prueba

## Casos de prueba

Un caso de prueba es un documento escrito que proporciona información escrita sobre qué y cómo probar

### Características de un buen caso de prueba

**No asumir:** No asumir la funcionalidad y las características de la aplicación mientras se prepara el caso de prueba. Se debe ser fiel a los documentos de especificación y ante cualquier duda, hay que consultar.

**Asegurar la mayor cobertura posible:** Escribir casos de prueba para todos los requisitos especificados.

**Autonomía**: El caso de prueba debe generar los mismos resultados siempre, sin importar quien lo pruebe.

**Evitar la repetición de casos de prueba:** Si se necesita un caso de prueba para ejecutar otro, indicar el caso de prueba por su ID.

**Deben ser simples**: Se deben crear casos de prueba que sean lo más simples posibles ya que otra persona que no sea el autor puede ejecutarlos. Utilizar un lenguaje asertivo para facilitar la comprensión y que la ejecución sea más rápida.

**El título debe ser fuerte**: Solo leyendo el título, cualquier probador debería comprender el objetivo del caso de prueba.

**Tener en cuenta al usuario final**: El objetivo final es crear casos de prueba que cumplan con los requisitos del cliente y que sean fáciles de usar.

### ¿Qué debe contener un caso de prueba?

**Identificador**: Puede ser numérico o alfanumérico. La mayoría de herramientas lo generan automáticamente.

**Nombre del caso de prueba (conciso)**: Se debe utilizar una nomenclatura que esté definida, pero, si no existe, lo recomendable es incluir el nombre de módulo al que corresponde el caso de prueba.

**Descripción**: Debe decir qué se va a probar, el ambiente de pruebas y los datos necesarios para ejecutarlo.

**Precondición**: Asunción que debe cumplirse antes de ejecutar el caso de prueba.

**Pasos:** Son las acciones que se deben realizar para obtener los resultados.

**Resultados esperados:** Es lo que le indica al probador cuál debería ser la experiencia luego de ejecutar los pasos y determinar si el test falló o no.

## Testing Negativo y Positivo

**Testing positivo:** Son aquellos casos de prueba que validan el flujo normal de un sistema bajo prueba. Es decir, flujos que están relacionados a los requisitos funcionales del sistema bajo prueba.

**Testing negativo:** Son aquellos casos de prueba que validan flujos no contemplados dentro de los requisitos de un sistema bajo prueba.

## Happy Path

Es el único camino con el que se prueba una aplicación a través de escenarios de prueba cuidadosamente diseñados, que deberían recorrer el mismo flujo que realiza un usuario final cuando usa la aplicación de manera regular. Generalmente es la primera forma de prueba que se realiza en una aplicación y se incluye en la categoría de prueba positiva. Su propósito no es encontrar defectos, sino ver que un producto o procedimiento funcione como ha sido diseñado.

#### Ventajas

* Se utiliza para conocer los estándares básicos de la aplicación. Es la primera prueba que se realiza
* Se utiliza para determinar la estabilidad de la aplicación antes de comenzar con otros niveles de prueba
* Ayuda a identificar cualquier problema en una etapa temprana y ahorrar esfuerzos posteriores

#### Desventajas

* No garantiza la calidad del producto porque el proceso solo utiliza escenarios de prueba positivos
* Encontrar este camino único requiere un gran conocimiento del uso de la aplicación y necesidades del cliente.

## Caso de uso vs. Caso de prueba

### ¿Qué es un caso de uso?

Un caso de uso cuenta la historia de cómo un usuario interactúa con un sistema de software para lograr o abandonar un objetivo. Cada caso de uso puede contener múltiples rutas que el usuario sigue, estos caminos son denominados escenario de caso de uso.

### ¿Qué es un caso de prueba?

Un caso de prueba cubre el software más en profundidad y con más detalle que un caso de uso. Estos incluyen todas las funciones que el programa es capaz de realizar y deben tener en cuenta el uso de todo tipo de datos de entrada/salida, cada comportamiento esperado y todos los elementos de diseño.

### ¿Cómo combinamos casos de uso con casos de prueba?

Se puede comenzar escribiendo casos de prueba para el “escenario principal” primero y luego escribirlos para “escenarios alternativos”. Es decir, se escribe uno o más casos de prueba por cada escenario de caso de uso. Los “pasos” de los casos de prueba se obtienen de la secuencia normal o alternativa detallada en los casos de uso.

Tanto el “nombre” como las “precondiciones” del caso de prueba se pueden basar directamente en los mismos campos que existen en el caso de uso. Para el “resultado esperado” de los casos de prueba se debe tener en cuenta la secuencia normal o alternativa y las poscondiciones del caso de uso.

La capacidad para crear casos de prueba a partir de los casos de uso y hacer la traza de unos a otros es una habilidad vital para asegurar un producto de calidad.

### ¿Qué son las pruebas de casos de uso?

Es una técnica de caja negra donde se verifica si la ruta utilizada por el usuario está funcionando según lo esperado o no. Se pueden crear uno o más casos de prueba para cada comportamiento detallado en los casos de uso —comportamiento básico o normal, excepcionales o alternativos y de tratamiento de errores—.

La cobertura se mide de la siguiente manera:

Tener en cuenta lo siguiente cuando se utiliza esta técnica de generación de pruebas a partir de casos de uso:

* Solo con las pruebas de casos de uso no se puede decidir la calidad del software.
* Incluso si es un tipo de prueba de extremo a extremo, no garantizará la cobertura completa de la aplicación del usuario.
* Los defectos pueden ser descubiertos posteriormente durante las pruebas de integración.

Clase 7: Técnicas de prueba

## Técnicas de Prueba

El objetivo de una técnica de prueba es ayudar a identificar las condiciones, los casos y los datos de prueba.

La elección de la técnica de prueba a utilizar depende de los siguientes factores:

* Tipo y complejidad del componente o sistema
* Estándares de regulación
* Requisitos del cliente o contractuales
* Clases y niveles de riesgo
* Objetivo de la prueba
* Documentación disponible
* Conocimientos y competencias del probador
* Modelo del ciclo de vida del software
* Tiempo y presupuesto

### Clasificación de las técnicas de prueba

#### Técnicas de caja negra

Se basan en el comportamiento extraído del análisis de los documentos que son base de prueba (documentos de requisitos formales, casos de uso, historias de usuario, etc). Son aplicables tanto para pruebas funcionales como no funcionales. Se concentran en las entradas y salidas sin tener en cuenta la estructura interna.

#### Técnicas de caja blanca

Se basan en la estructura extraída de los documentos de arquitectura, diseño detallado, estructura interna o código del sistema. Se concentran en el procesamiento dentro del objeto de prueba.

#### Técnicas basadas en la experiencia

Aprovechan el conocimiento de desarrolladores, probadores y usuarios para diseñar, implementar y ejecutar las pruebas.

### Técnicas de caja negra

#### Partición de equivalencia

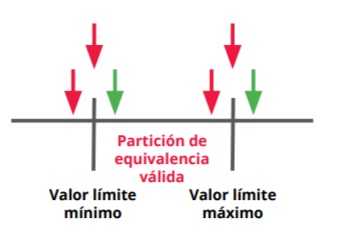
En esta técnica se dividen los datos en particiones conocidas como clases de equivalencia donde cada miembro de estas clases o particiones es procesado de la misma manera. Las características de esta técnica son:

* La “partición de equivalencia válida” contiene valores que son aceptados por el componente o sistema.
* La “partición de equivalencia no válida” contiene valores que son rechazados por el componente o sistema.
* Se pueden dividir las particiones en subparticiones.
* Cada valor pertenece a solo una partición de equivalencia.
* Las particiones de equivalencia no válidas deben probarse en forma individual para evitar el enmascaramiento de fallos.
* La cobertura se mide de la siguiente manera:

#### Análisis de valores límites

Es una extensión de la técnica de partición de equivalencia que solo se puede usar cuando la partición está ordenada, y consiste en datos numéricos o secuenciales.

* Se deben identificar los valores límites mínimo y máximo (o valores inicial y final).
* Se pueden utilizar 2 o 3 valores límites.
* Para 2 valores límites se toma el valor que marca el límite (como valor que corresponde a la partición válida), y el valor anterior o posterior que corresponda a la partición de equivalencia inválida.
* Para 3 valores límites se toma el valor que marca el límite, un valor anterior y otro posterior a ese límite.
* La cobertura se mide de la siguiente manera:



#### Tabla de decisión

Esta técnica se utiliza para pruebas combinatorias, formadas por reglas de negocio complejas que un sistema debe implementar. Las características de esta técnica son:

* Se deben identificar las condiciones (entradas) y las acciones resultantes (salidas). Estas conforman las filas de la tabla.
* Las columnas de la tabla corresponden a reglas de decisión. Cada columna forma una combinación única de condiciones y la ejecución de acciones asociadas a esa regla.
* Los valores de las condiciones y acciones pueden ser valores booleanos, discretos, numéricos o intervalos de números.
* Ayuda a identificar todas las combinaciones importantes de condiciones y a encontrar cualquier desfase en los requisitos.
* La cobertura se mide de la siguiente manera:

### Técnicas basadas en la experiencia

#### Predicción de errores

Esta técnica se utiliza para anticipar la ocurrencia de equivocaciones, defectos y fallos basados en el conocimiento del probador. Se crea una lista teniendo en cuenta:

* Cómo ha funcionado la aplicación en el pasado.
* Equivocaciones comunes en los desarrolladores.
* Fallos en aplicaciones relacionadas.

En base a esa lista se diseñan pruebas que expongan esos fallos y defectos

### Prueba exploratoria

En esta técnica se diseñan, ejecutan, registran y evalúan de forma dinámica pruebas informales durante la ejecución de la prueba.

Los resultados de estas pruebas se utilizan para aprender más sobre el funcionamiento del componente o sistema.

Generalmente se utilizan para complementar otras técnicas formales o cuando las especificaciones son escasas, inadecuadas o con restricciones de tiempo.

### Prueba basada en listas de comprobación

En esta técnica se diseñan, implementan y ejecutan casos de prueba que cubren las condiciones que se encuentran en una lista de comprobación definida.

Se crean basadas en la experiencia y conocimiento de lo que el probador cree que es importante para el usuario y se utilizan debido a la falta de casos de prueba detallados.

Durante la ejecución puede haber cierta variabilidad, dependiendo de quién ejecuta la prueba y condiciones del contexto. Esto da lugar a una mayor cobertura. Se utiliza tanto en pruebas funcionales como no funcionales.

Clase 8: Implementación y Ejecución de la prueba

# Módulo 4: Implementación de la prueba

## Ejecución de los casos de prueba

Durante la ejecución de la prueba, los juegos de pruebas se realizan de acuerdo a un calendario de ejecución de la prueba, que incluye diferentes actividades principales. Para conocerlas, pasar con el mouse por encima de cada uno de los iconos que se encuentran a continuación.

1. Registrar los identificadores y las versiones de los elementos u objetos de prueba, las herramientas de prueba y los productos de prueba.
2. Ejecutar pruebas de forma manual o utilizando herramientas de ejecución de pruebas.
3. Comparar resultados reales con resultados esperados.
4. Analizar las anomalías para establecer sus causas probables.
5. Informar sobre los defectos en función de los fallos observados.
6. Registrar el resultado de la ejecución de la prueba.
7. Repetir las actividades de prueba, ya sea como resultado de una acción tomada para una anomalía o como parte de la prueba planificada.

## Creación de suites

Durante el proceso de desarrollo de software el momento más propenso para la inclusión involuntaria de fallas suele ser cuando se introducen nuevas funcionalidades en la aplicación. Para atacar el inconveniente vemos dos tipos de conjuntos o juegos de caso de prueba muy utilizados al momento del despliegue de nuevas funcionalidades estas, son una colección de caso de prueba con un fin específico.

El primer conjunto se conoce como pruebas de humo aquellas que cubren la funcionalidad principal de un componente o sistema. El objetivo es asegurar que las funciones cruciales de un programa funcionan, pero sin preocuparse por los detalles finos. Así, con pruebas sencillas y que demandan poco tiempo verificamos que funcionan correctamente ciertos caminos de la aplicación. Generalmente se elige sólo un conjunto de funcionalidades significativas de la aplicación.

El segundo conjunto o colección de casos de prueba son las de regresión y nos permiten asegurarnos que los cambios no han dañado las interfaces, los componentes o los sistemas existentes. Además, se buscan cambios no deseados en el comportamiento que resulten de cambios en el software o en el entorno. Dentro de un proyecto de automatización lo ideal es comenzar con las pruebas de regresión ya que esta se ejecuta muchas veces y generalmente evoluciona lentamente.

La correcta aplicación de un proceso de pruebas durante el despliegue implicaría, primero, ejecutar las pruebas de humo y una vez confirmada la ejecución exitosa de estas pruebas estamos en condiciones de regresionar nuestro sistema bajo prueba.

¿Por qué es importante ejecutar estas pruebas? Primero, las pruebas de humo nos ayudan a confirmar que luego de un despliegue las funcionalidades principales no sufrieron fallas. Por otro lado, las pruebas de regresión, terminan de confirmar que todo lo que antes del despliegue funcionaba sigue funcionando la misma forma.

Es probable que estas pruebas se complementen con pruebas específicas relacionadas a las nuevas funcionalidades que se desplegarán.

Clase 10: Pruebas Estáticas y Dinámicas

# Módulo 5: Análisis de la Prueba

## Pruebas Estáticas y Dinámicas

Las pruebas estáticas y dinámicas tienen el objetivo de proporcionar una evaluación de calidad de los productos de trabajo e identificar defectos en forma temprana.

### Pruebas Estáticas

La prueba estática se basa en la evaluación manual de los productos de trabajo (revisiones) o en la evaluación basada en herramientas del código u otros productos de trabajo (análisis estático).

Algunos de los productos de trabajo a revisar o analizar son:

* Especificaciones, requisitos de negocio, funcionales y de seguridad.
* Épicas, historias de usuarios y criterios de aceptación.
* Especificaciones de arquitectura y diseño.
* Código.
* Productos de prueba: planes, casos, procedimientos y guiones de prueba.
* Manuales de usuario.
* Contratos, planes de proyecto, calendarios y presupuestos.

#### Ventajas de las pruebas estáticas

Cuando se aplica al principio del ciclo de vida del desarrollo del software, la prueba estática permite la detección temprana de defectos. Esto genera una reducción de costos y tiempo de desarrollo y prueba.

Por el contrario, si el defecto se encuentra luego de las pruebas dinámicas, solucionarlo va a requerir el cambio de código, realizar una prueba de confirmación y luego incluir el mismo en pruebas de regresión, además de los cambios de toda la documentación asociada.

#### Defectos encontrados en las pruebas estáticas

Algunos de los defectos encontrados con pruebas estáticas que son más fáciles y económicos de detectar y corregir son:

* Defectos en los requisitos (inconsistencias, ambigüedades, etc.).
* Defectos de diseño (estructura de base de datos ineficiente, alto acoplamiento, etc.).
* Defectos de codificación (variables con valores no definidos, código inalcanzable o duplicado, etc.).
* Desviaciones con respecto a estándares (falta de uso de estándares de codificación).
* Especificaciones de interfaz incorrectas (unidades de medida diferente, etc.).
* Vulnerabilidades de seguridad (susceptibilidad a desbordamiento de la memoria intermedia).
* Diferencias o inexactitudes en la trazabilidad o cobertura de la base de prueba (falta de pruebas para un criterio de aceptación).
* Defectos de mantenibilidad (mala reutilización de componentes, modularización inadecuada, etc.).

### Pruebas Dinámicas

Las pruebas dinámicas requieren la ejecución del software, componente o sistema. Se complementan con las pruebas estáticas debido a que encuentra diferentes tipos de defectos. Para la generación de casos de prueba se utilizan diferentes técnicas de caja negra, caja blanca o basadas en la experiencia de usuario.

Durante las pruebas dinámicas se ejecuta el software utilizando un conjunto de valores de entrada y su resultado se analiza y compara con el resultado esperado.

Las fallas más comunes encontradas con este tipo de pruebas son:

* Fallas de funcionalidad.
* Fallas de interacción entre módulos.
* Fallas de rendimiento y seguridad.

|  |  |
| --- | --- |
| Prueba estática | Prueba dinámica |
| Detecta los defectos en productos de trabajo. | Detecta los defectos y fallas cuando se ejecuta el software. |
| Se centra en mejorar la consistencia y la calidad de los productos de trabajo. | Se centra en los comportamientos visibles desde el exterior. |
| El costo de solucionar un defecto es menor | El costo de solucionar un defecto es mayor. |

## Proceso de Revisión

Las revisiones consisten en examinar cuidadosamente un producto de trabajo con el principal objetivo de encontrar y remover errores. Pueden ser realizadas por una o más personas.

Las revisiones pueden ser:

* **Revisiones formales**: Tienen roles definidos, siguen un proceso establecido y deben ser documentadas.
* **Revisiones Informales**: No siguen un proceso definido y no son documentadas formalmente.

El grado de formalidad del proceso de revisión está relacionado con factores, como el modelo del ciclo de vida del desarrollo del software, la madurez del proceso de desarrollo, la complejidad del producto del trabajo que se debe revisar, cualquier requisito legal y/o la necesidad de un rastro de auditoría.

### Roles

* **Autor**: Creador del producto de trabajo bajo revisión y quien corrige los defectos, en caso de ser necesario.
* **Dirección**: Planifica y controla las revisiones.
* **Facilitador/Moderador**: Asegura el funcionamiento efectivo de las reuniones de revisiones y la modera.
* **Líder de revisión:** Asume la responsabilidad general de la revisión; decide quiénes estarán involucrados y organiza cuándo y dónde se llevará a cabo.
* **Revisores**: Identifican posibles defectos en el producto de trabajo bajo revisión; pueden representar diferentes perspectivas —por ejemplo, probador, programador, usuarios, operador, analista de negocio, experto en usabilidad—.
* **Escriba**: Recopila los posibles defectos encontrados, puntos abiertos y decisiones en las revisiones.

### Tipos

* **Informal**: No tiene un proceso formal documentado ni es necesaria una reunión de revisión, puede ser realizada entre colegas. El uso de listas de comprobación o checklist es opcional. La documentación de los defectos no es obligatoria.
* **Guiada | Walkthrough**: Dirigida por el autor del producto de trabajo, el escriba es obligatorio. El uso de listas de comprobación y la preparación individual previa son opcionales. Puede variar de informal a formal. Puede o no haber documentación de defectos potenciales.
* **Técnica**: Se realiza entre pares técnicos del autor, si hay una reunión debe haber una preparación individual previa. Tienen que existir un facilitador y un escriba obligatoriamente, quien idealmente no será el autor. Se elaboran registros de defectos potenciales e informes de revisión.
* **Inspecciones**: Los resultados se documentan formalmente. Requieren preparación individual, tienen roles definidos: autor, director, facilitador, escriba, revisores y líder de revisión, pueden incluir también un lector dedicado. El autor no puede actuar como facilitador, líder de revisión, lector o escriba. Se hace uso de listas de comprobación. Se elaboran registros de defectos potenciales e informes de revisión.

### Técnicas

* **Ad hoc:** Los revisores leen el producto de trabajo de forma secuencial y a medida que van identificando los defectos, los van documentando, recibiendo poca o ninguna orientación en el proceso.
* **Basada en escenarios y ensayos:** Fundada en el uso esperado del producto de trabajo descrito en un documento, por ejemplo, en un caso de uso.
* **Basada en listas de comprobación**: Los revisores detectan defectos a partir de un conjunto de preguntas basadas en defectos potenciales, producto de la experiencia del autor de la lista.
* **Basada en roles:** Los revisores evalúan el producto de trabajo desde la perspectiva de usuarios experimentados, inexpertos, adultos, niños o roles específicos en la organización, como un administrador de usuarios, de sistemas o un probador del rendimiento.
* **Basada en perspectiva**: Esta técnica es similar a una revisión basada en roles, los revisores adoptan los diferentes puntos de vista del usuario final, del personal de marketing, del diseñador, del probador o del personal de operaciones.

### Actividades

* **Planificar**: Definir el alcance, establecer objetivos, roles, tiempo y plazos. Definir y comprobar el cumplimiento de criterios de entrada y salida para revisiones más formales.
* **Iniciar revisión**: Distribuir el material e instruir a los participantes.
* **Revisión individual (preparación individual**): Revisión del material y tomar notas de los defectos encontrados.
* **Comunicar y analizar**: Comunicar defectos a los responsables.
* **Corregir e informar**: Comunicar los defectos potenciales encontrados en una reunión de revisión. Elaborar informes de hallazgos, documentar las características de calidad y comprobar criterios de salida.

## Requerimientos

Un requisito define las funciones, capacidades o atributos intrínsecos de un sistema de software, es decir, describe **cómo debe comportarse un sistema.** Para decir que un sistema tiene calidad deben cumplirse los requisitos funcionales y no funcionales.

### Requisitos funcionales

Definen lo que un sistema permite hacer desde el punto de vista del usuario. Estos requisitos deben estar especificados de manera explícita.

Ejemplo: El campo de monto acepta únicamente valores numéricos con dos decimales (pruebas funcionales y de sistema).

### Requisitos no funcionales

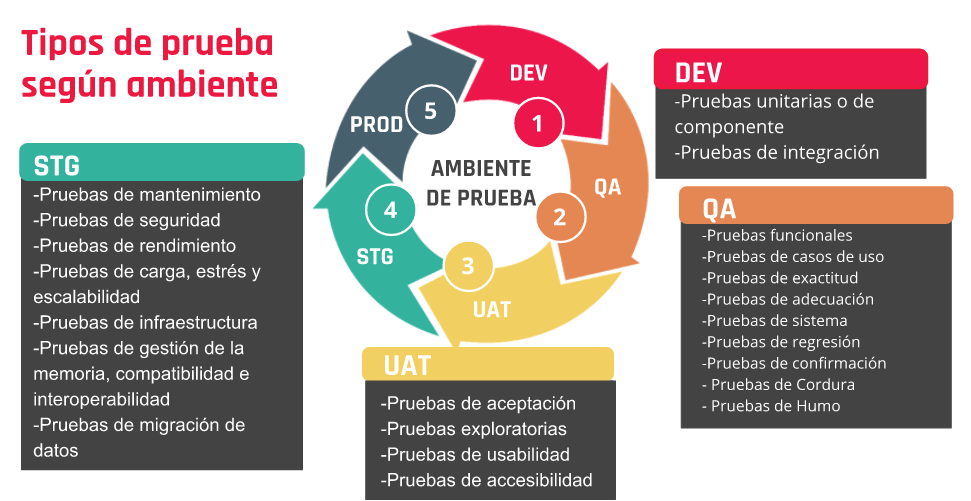
Definen condiciones de funcionamiento del sistema en el ambiente operacional. Ejemplos:

* Requisito de usabilidad: La usabilidad se define como el esfuerzo que necesita hacer un usuario para aprender, usar, ingresar datos e interpretar los resultados obtenidos de un software de aplicación (pruebas de usabilidad).
* Requisito de eficiencia: Relacionado con el desempeño en cuanto al tiempo de respuesta, número de operaciones por segundo, entre otras mediciones, así como consumo de recursos de memoria, procesador y espacio en disco o red (pruebas de rendimiento, pruebas de carga, estrés y escalabilidad, pruebas de gestión de la memoria, compatibilidad e interoperabilidad).
* Requisito de disponibilidad: Disposición del sistema para prestar un servicio correctamente (pruebas de disponibilidad).
* Requisito de confiabilidad: Continuidad del servicio prestado por el sistema (pruebas de seguridad).
* Requisito de integridad: Ausencia de alteraciones inadecuadas al sistema (pruebas de seguridad, pruebas de integridad).
* Requisito de mantenibilidad: Posibilidad de realizar modificaciones o reparaciones a un proceso sin afectar la continuidad del servicio (pruebas de mantenimiento y de regresión).

Clase 11: Organización de la prueba

# Módulo 6: Planificación de la prueba

## Entornos de prueba



### DEV

#### Pruebas unitarias o de componente

También se conocen como pruebas de módulo. Se centra en los componentes que se pueden probar por separado. Tiene como objetivo encontrar defectos en el componente y verificar que los comportamientos funcionales y no funcionales del componente son los diseñados y especificados.

#### Pruebas de Integración

Se centra en las interacciones entre componentes o sistemas. Los objetivos de la prueba de integración incluyen encontrar defectos en las propias interfaces o dentro de los componentes o sistemas y verificar que los comportamientos funcionales y no funcionales de las interfaces sean los diseñados y especificados.

### QA

#### Pruebas funcionales

Incluye pruebas que evalúan las funciones que el sistema debe realizar. Los requisitos funcionales pueden estar descritos en productos de trabajo tales como especificaciones de requisitos de negocio, épicas, historias de usuario, casos de uso y especificaciones funcionales. También pueden estar sin documentar.

#### Pruebas de casos de uso

Proporcionan pruebas transaccionales, basadas en escenarios, que deberían emular el uso del sistema.

#### Pruebas de exactitud

Comprenden el cumplimiento por parte de la aplicación de los requisitos especificados o implícitos y también puede abarcar la exactitud de cálculo.

#### Pruebas de adecuación

Implican evaluar y validar la eficiencia de un conjunto de funciones para la consecución de las tareas especificadas previstas. Estas pruebas pueden basarse en casos de uso.

#### Pruebas de sistema

Se centra en el comportamiento y las capacidades de todo un sistema o producto, a menudo teniendo en cuenta las tareas extremo a extremo que el sistema puede realizar y los comportamientos no funcionales que exhibe mientras realiza esas tareas.

#### Pruebas de regresión

Implican la realización de pruebas para detectar efectos secundarios no deseados, luego de cambios hechos en una parte del código que puedan afectar accidentalmente el comportamiento de otras partes del código.

#### Pruebas de confirmación

Consiste en volver a ejecutar los pasos para reproducir el fallo o los fallos causados por un defecto en la nueva versión de software, una vez corregido el defecto, para así confirmar que el defecto original se ha solucionado satisfactoriamente o detectar efectos secundarios no deseados.

#### Pruebas de Cordura

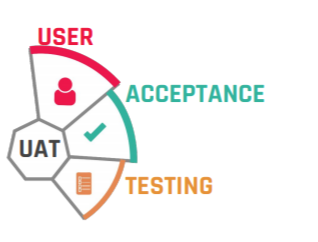
Es una prueba de regresión acotada que se centra en una o unas pocas áreas de funcionalidad. Se utiliza para determinar si una pequeña sección de la aplicación sigue funcionando después de un cambio menor.

#### Pruebas de Humo

Se lleva a cabo un smoke test para asegurar si las funciones más importantes de un programa están trabajando correctamente, pero sin molestarse con los detalles más finos.

### UAT

#### Pruebas de aceptación

Se centra normalmente en el comportamiento y las capacidades de todo un sistema o producto. Además, pueden producir información para evaluar el grado de preparación del sistema para su despliegue y uso por parte del cliente (usuario final).

#### Pruebas exploratorias

Se diseñan, ejecutan, registran y evalúan de forma dinámica pruebas informales (no predefinidas) durante la ejecución de la prueba. Los resultados de la prueba se utilizan con el objetivo de aprender más sobre el componente o sistema y crear pruebas para las áreas que pueden necesitar ser probadas con mayor intensidad.

#### Pruebas de usabilidad

Evalúan la facilidad con la que los usuarios pueden utilizar o aprender a utilizar el sistema para lograr un objetivo específico en un contexto dado.

#### Pruebas de accesibilidad

Incluyen y evalúan la accesibilidad que presenta un software para aquellos con necesidades particulares o restricciones para su uso. Esto incluye a aquellos usuarios con discapacidades.

### STG

#### Pruebas de mantenimiento

Se centra en probar los cambios en el sistema, así como en probar las piezas no modificadas que podrían haberse visto afectadas por los cambios. El mantenimiento puede incluir lanzamientos planificados y no planificados.

#### Pruebas de seguridad

Las pruebas de seguridad se podrían definir como el conjunto de actividades que se llevan a cabo para encontrar fallas y vulnerabilidades en el sistema, buscando disminuir el impacto de ataques y pérdida de información importante.

#### Pruebas de rendimiento

Se implementan y se ejecutan para evaluar las características relacionadas con el rendimiento del destino de la prueba, como los perfiles de tiempo, el flujo de ejecución, los tiempos de respuesta y la fiabilidad y los límites operativos. También se pueden realizar en STG.

#### Pruebas de carga, estrés y escalabilidad

Una prueba de carga garantiza que un sistema pueda controlar un volumen de tráfico esperado. Una prueba de estrés es en la que se somete al sistema a condiciones de uso extremas para garantizar su robustez y confiabilidad. Las pruebas de escalabilidad garantizan la escalabilidad de un sistema, es decir, que pueda soportar el incremento de demanda en la operación. También se pueden realizar en QA encontrando el correspondiente escalar con respecto a un ambiente de PROD.

#### Pruebas de infraestructura

Incluyen todos los sistemas informáticos internos, los dispositivos externos asociados, las redes de Internet, la nube y las pruebas de virtualización.

#### Pruebas de gestión de la memoria

Evalúan el estado y la integridad de la memoria del sistema para identificar problemas potenciales.

#### Pruebas de compatibilidad

Incluyen las pruebas para comprobar que el sistema es compatible con todos los navegadores de Internet y todos los sistemas operativos del mercado.

#### Pruebas de interoperabilidad

Se refieren a aquellas donde se realiza la evaluación de la correcta integración entre distintos aplicativos, sistemas, servicios o procesos que conforman una plataforma o solución tecnológica.

#### Pruebas de migración de datos

Incluyen las pruebas realizadas al transferir datos entre tipos de dispositivos de almacenamiento, formatos o sistemas de cómputo.

## Métricas y reportes

Las métricas de pruebas pueden clasificarse en las siguientes categorías:

Métricas del proyecto: miden el progreso hacia los criterios de salida previstos del proyecto, tales como el porcentaje de casos de prueba ejecutados, pasados y fallidos.

Métricas del producto: miden algunos atributos del producto, tales como el alcance en el que ha sido probado o la densidad del defecto.

Métricas del proceso: miden la capacidad del proceso de pruebas de desarrollo, tales como el porcentaje de defectos detectados por las pruebas.

Métricas de las personas: miden la capacidad de los individuos o los grupos, tales como la implementación de casos de prueba dentro de un calendario dado.

El uso de métricas permite a los probadores comunicar resultados de una manera consistente y llevar a cabo un seguimiento coherente del progreso a lo largo del tiempo.

Existen cinco dimensiones primarias sobre las que se monitoriza el progreso de las pruebas:

* Riesgos (de la calidad) del producto
* Defectos
* Pruebas
* Cobertura
* Confianza

Durante la monitorización y el control de prueba, el jefe de prueba emite periódicamente informes de avance de prueba para los implicados. Los informes de avance típicos pueden incluir:

* El estado de las actividades de prueba y el avance con respecto al plan de prueba.
* Factores que impiden el avance.
* Pruebas previstas para el próximo período objeto del informe.
* La calidad del objeto de prueba.

Cuando se alcanzan los criterios de salida, el jefe de prueba emite un informe del resumen de la prueba. Los informes de resumen de la prueba típicos pueden incluir:

* Resumen de la prueba realizada
* Información sobre lo ocurrido durante un período de prueba.
* Desviaciones con respecto al plan, incluye los cambios en el calendario, la duración o el esfuerzo de las actividades de prueba.
* Estado de la prueba y calidad del producto con respecto a los criterios de salida o definición de hecho.
* Factores que han bloqueado o continúan bloqueando el avance.
* Métricas de defectos, casos de prueba, cobertura de la prueba, avance de la actividad y consumo de recursos.
* Riesgos residuales.
* Productos de trabajo de la prueba reutilizables desarrollados.

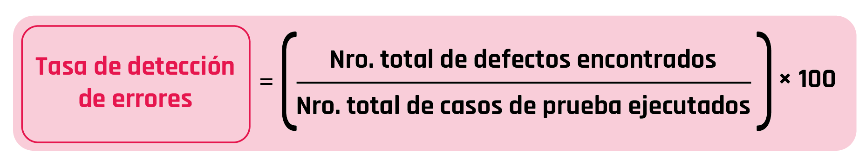
El contenido específico de los informes de prueba debería variar en función del contexto del proyecto y del público al cual está destinado dicho informe. El tipo y cantidad de información que se debe incluir para una audiencia técnica o un equipo de prueba pueden ser diferentes de las que se incluirán en un informe de resumen ejecutivo.

En el desarrollo con metodologías ágiles el informe de avance de la prueba puede estar incorporado en los tableros de tareas, resúmenes de defectos y gráficos de trabajo pendiente, los cuales pueden ser discutidos durante las ceremonias vinculadas a esta metodología.

### Métrica del producto

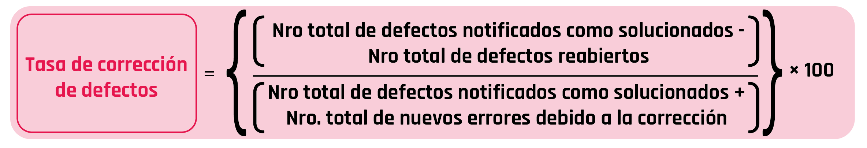
#### Tasa de detección de errores

Es para determinar la efectividad de los casos de prueba.



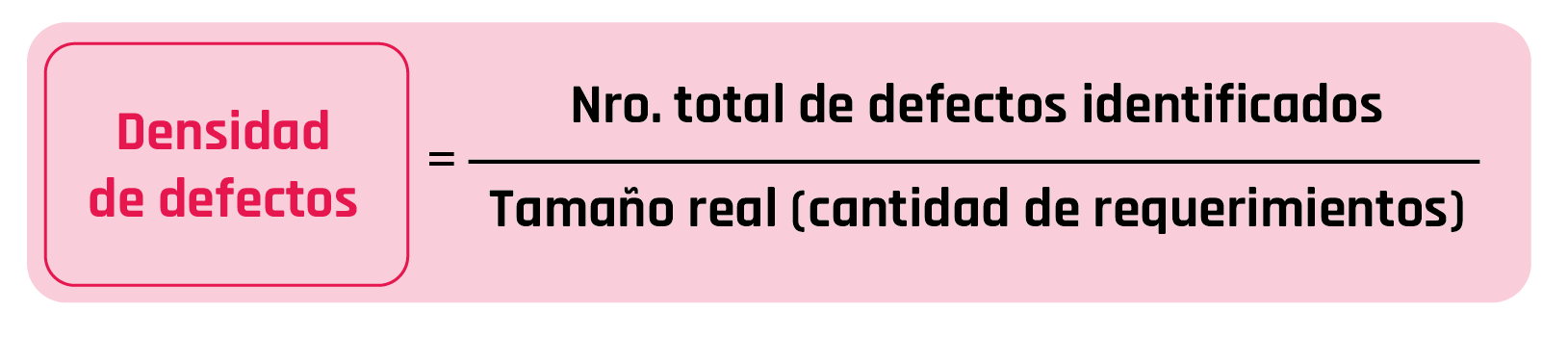
#### Tasa de corrección de defectos

Ayuda a conocer la calidad del software en términos de reparación de defectos.



#### Densidad de defectos

Se define como la relación entre defectos y requisitos.



#### Fugas por defecto

Se utiliza para revisar la eficiencia del proceso de prueba antes de UAT.



#### Eficiencia de eliminación de defectos

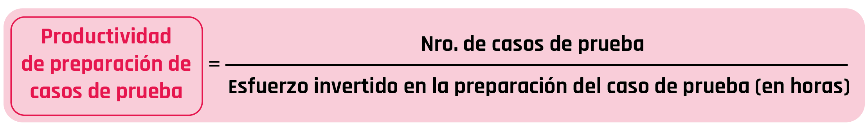
Permite comparar la eficiencia general de eliminación de defectos (defectos encontrados antes y después de la entrega).



### Métricas de proceso

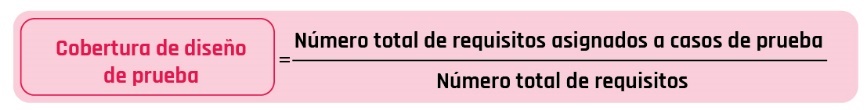
#### Productividad en la preparación de casos de prueba

Se utiliza para calcular el número de casos de prueba preparados y el esfuerzo invertido en la preparación de casos de prueba.



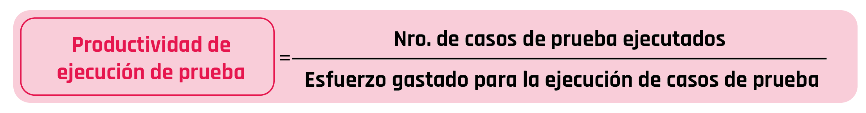
#### Cobertura de diseño de prueba

Ayuda a medir el porcentaje de cobertura de casos de prueba frente al número de requisitos.



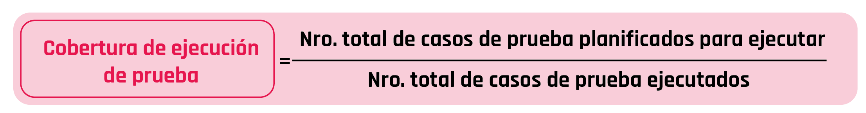
#### Productividad de ejecución de prueba

Determina el número de casos de prueba que se pueden ejecutar por hora.



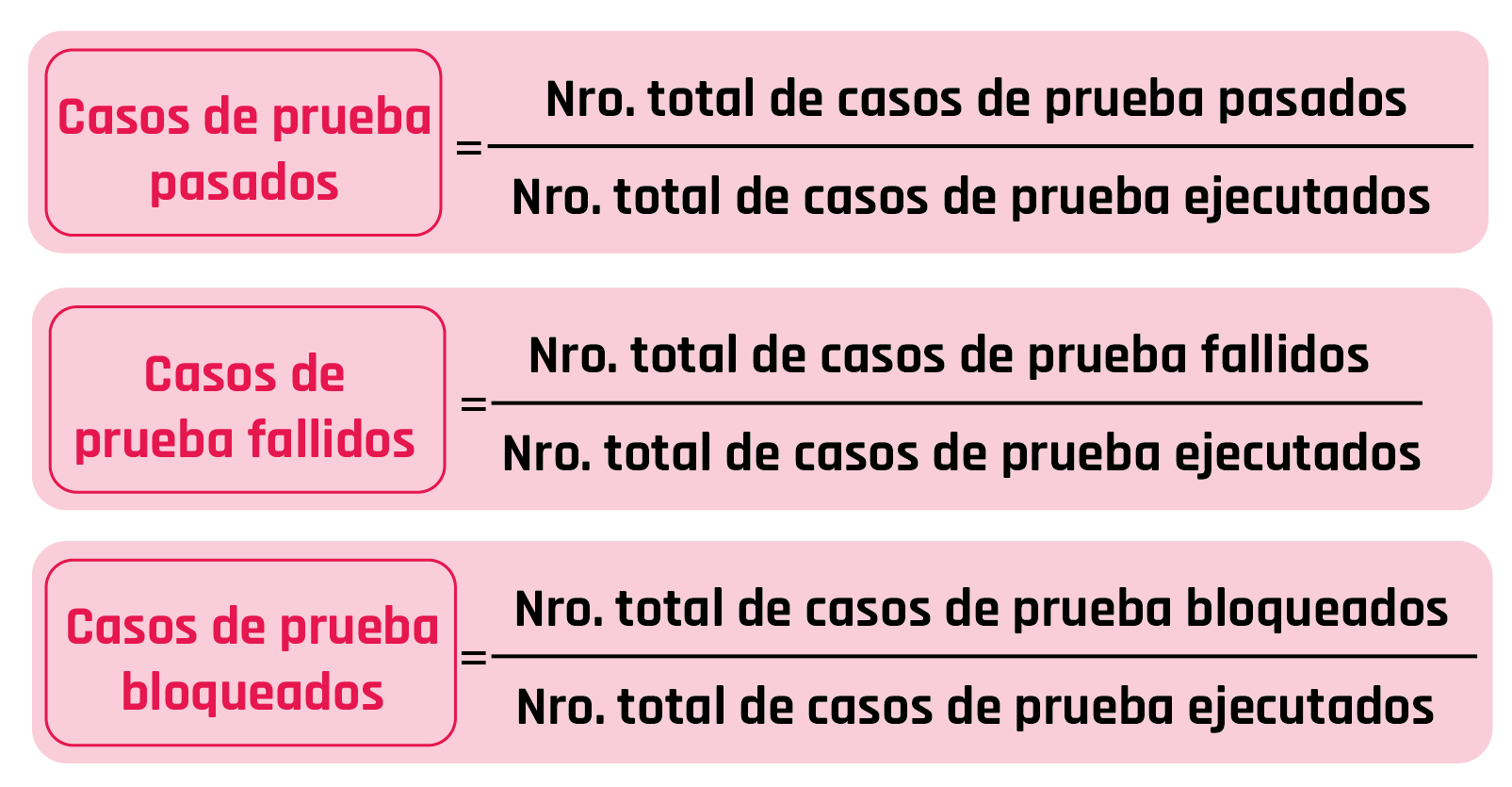
#### Cobertura de ejecución de pruebas

Es para medir el número de casos de prueba ejecutados contra el número de casos de prueba planteados.



#### Casos de prueba pasados, fallidos y bloqueados

Es para medir el porcentaje de los casos de prueba pasados, fallidos y bloqueados.



Clase 14: Introducción a la prueba de Componente

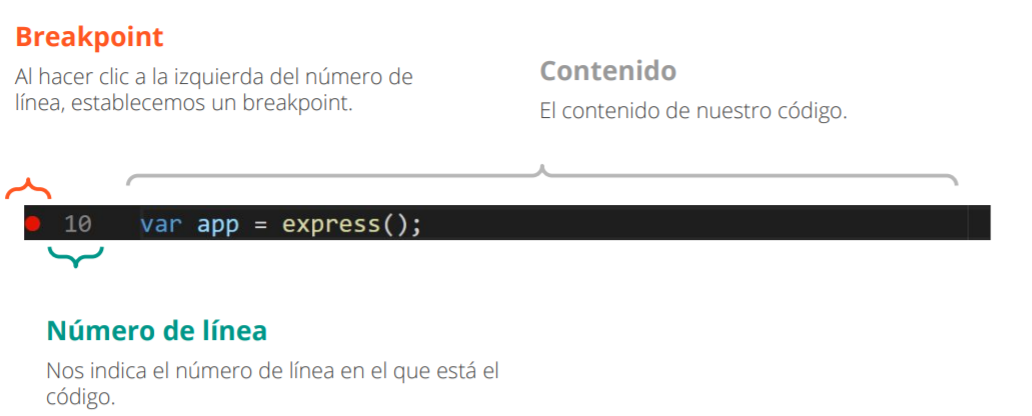
# Módulo 7: Introducción a TDD

## Debug

Llamamos debuggear o depurar al proceso de encontrar, analizar y remover las causas de fallos en en el software. Lo que se realiza es la ejecución paso a paso de cada instrucción del programa para analizar las variables y sus valores.

### Breakpoints

Un breakpoint es una pausa en nuestro código para entender en qué estado están nuestras variables.



#### Control de variables

Como podemos ver, en el breakpoint nuestra variable vale “2” y eso está reflejado en el menú de debug a la izquierda.



### Interfaz

Cuando abrimos el menú de debug nos despliega el siguiente menú:



|  |  |
| --- | --- |
| Icono | Función |
|  | **Continue**: Continua con la ejecución del código hasta el siguiente breakpoint |
|  | **Step over:** Ejecuta la siguiente línea de código. Si es una función, la ejecuta y retorna el resultado. |
|  | **Step into:** Ejecuta la siguiente línea de código, pero si es una función “entra” a ejecutarla línea por línea. |
|  | **Step out**: Devuelve el debugger a la línea donde se llamó a la función. |
|  | **Restart**: Reinicia el debugger. |
|  | **Stop**: Frena el debug. |

Tabla 2: Controles

### Debug desde la consola de Chrome

Es posible depurar código JavaScript directamente desde la consola de Chrome, para esto se deben seguir los siguientes pasos:

1. Acceder a la consola de Chrome presionando la tecla F12, o desde la opción de menú “Herramientas del desarrollador” que se encuentra en “Más herramientas”.
2. En “Sources”, seleccionar el archivo a depurar.
3. Marcar el breakpoint en la línea de código correspondiente.
4. Presionar F11 para recorrer linea por linea y F8 para recorrer de un breakpoint a otro.

### Debugging vs Testing

|  |  |
| --- | --- |
| Debugging | Testing |
| Proceso deductivo para corregir los errores encontrados durante las pruebas | Proceso donde se comprueba que el sistema o componente funcione de acuerdo a lo esperado, tiene como objetivo la búsqueda de errores. |
| Permite dar solución a la falla del código | Permite identificar la/s falla/s del código implementado |
| Es la investigación y detección de error | Es la visualización de errores |
| Realizado generalmente por el programador o el desarrollador, a excepción del código generado por el probador como parte de los scripts de prueba automáticos | Realizado generalmente por el probador |
| No se puede realizar sin el conocimiento de diseño adecuado | No es necesario tener conocimientos de diseño en el proceso de prueba |
| Lo realiza únicamente un interno. Generalmente un externo no puede depurar debido a que no debería tener acceso al código | Puede ser realizado tanto por personas internas como externas |
| Realizado en forma manual | Puede ser manual o automatizado |
| Basado en estrategias de debugging como depuración por fuerza bruta, bug-tracking, eliminación de causas, depuración automatizada utilizando herramientas y la mirada de un colega | Basado en diferentes niveles de prueba, es decir, pruebas de componente, pruebas de integración, pruebas del sistema, etc. |
| No es un aspecto del ciclo de vida del desarrollo de software, ocurre como consecuencia de las pruebas | Etapa del ciclo de vida del desarrollo de software (SDLC) |
| Busca hacer coincidir el síntoma con la causa, lo que conduce a la corrección del error | Compuesto por la validación y verificación del software |
| Comienza con la ejecución del código debido a un caso de prueba fallido | Iniciado ante de tener el código (testing estático) o después de que se escribe el código (testing dinámico) |

Tabla 3: Debugging vs Testing

### Prueba de componente o prueba unitaria (unit test)

La prueba de componente o prueba unitaria es la prueba de los componentes individuales de software. Son pequeños test creados específicamente para cubrir todos los requisitos del código y verificar sus resultados. Para generar estos test se utilizan técnicas de caja blanca.

Las pruebas unitarias son generalmente pruebas automatizadas escritas y ejecutadas por desarrolladores de software para garantizar que una sección de una aplicación —conocida como la "unidad"— cumpla con su diseño y se comporte según lo previsto.

El proceso general para la creación de estos unit test consta de tres partes:

1. Acuerdo o criterio de aceptación: donde se definen los requisitos que debe cumplir el código principal.
2. Escritura del test: el proceso de creación, donde se acumulan los resultados a analizar.
3. Confirmación: se considera el momento en que comprobamos si los resultados agrupados son correctos o incorrectos. Dependiendo del resultado, se valida y continúa, o se repara, de forma que el error desaparezca (debug).

#### ¿Qué es una “unidad”?

Una unidad puede ser casi cualquier parte del código que queremos que sea: una línea de código, un método o una clase. En general, cuanto más pequeño, mejor. Las pruebas más pequeñas brindan una vista mucho más granular de cómo se está desempeñando el código. También existe el aspecto práctico de que cuando se prueban unidades muy pequeñas, se pueden ejecutar rápidamente.

#### Framework para pruebas unitarias

Es una herramienta que proporciona un entorno para la prueba de unidades o componentes en el que un componente se puede probar de forma aislada o con adecuados stubs y drivers. También proporciona otro soporte para el desarrollador, como la capacidad de depuración.

Por ejemplo, para realizar la prueba de componente de código realizado en JavaScript se puede crear un framework basado en las siguientes herramientas:

* Mocha (https://mochajs.org/api/index.html)
* Chai (https://www.chaijs.com/guide/)

#### Desarrollo guiado por pruebas (TDD)

El desarrollo guiado por pruebas es una forma de desarrollar software donde se desarrollan los casos de prueba, generalmente automatizados, antes de que se desarrolle el software para ejecutar esos casos de prueba.

El desarrollo guiado por pruebas es altamente iterativo y se basa en ciclos de desarrollo de casos de prueba automatizados, luego se construyen e integran pequeños fragmentos de código, a continuación, se ejecuta la prueba de componente, se corrige cualquier cuestión y se refactoriza el código. Este proceso continúa hasta que el componente ha sido completamente construido y ha pasado toda la prueba de componente.

Clase 16: Prueba de Componente

## Prueba de componentes

### Generalidades

El objetivo principal es aislar cada unidad del sistema para identificar, analizar y corregir los defectos.

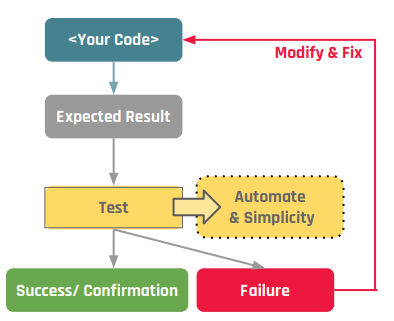
La prueba de componente, a menudo, se realiza de forma aislada del resto del sistema, dependiendo del modelo de ciclo de vida de desarrollo de software y del sistema, lo que puede requerir objetos simulados, virtualización de servicios, arneses, stubs y controladores.

Este tipo de pruebas puede cubrir:

* **La funcionalidad**: por ejemplo, la exactitud de los cálculos.
* **Las características no funcionales**: por ejemplo, la búsqueda de fugas de memoria
* **Las propiedades estructurales**: por ejemplo, pruebas de decisión.

En general, cuando no se sigue un enfoque TDD, el proceso es el siguiente:

1. Se crea el código del software.
2. Se definen los resultados esperados.
3. Se ejecuta el test.
   1. Si el test pasa, se confirma el resultado esperado.
   2. Si el test falla, se modifica el código para solucionar el defecto encontrado Lo ideal es automatizar los test para poder simplificar el proceso de prueba.



### Ventajas

* Reduce los defectos en las funciones recientemente desarrolladas o reduce los errores al cambiar la funcionalidad existente.
* En modelos de desarrollo incrementales e iterativos donde los cambios de código son continuos, la prueba de regresión de componente automatizada juega un papel clave en la construcción de la confianza en que los cambios no han dañado a los componentes existentes.
* Las pruebas unitarias inapropiadas harán que los defectos se propaguen hacia pruebas de nivel superior y esto conducirá a un alto costo de reparación de defectos durante las pruebas del sistema, las pruebas de integración e incluso las pruebas de aceptación de usuario. Si se realizan las pruebas unitarias adecuadas en el desarrollo inicial, al final se ahorra esfuerzo, tiempo y dinero.

### Frameworks

Las pruebas unitarias pueden ser de dos tipos:

Manuales: Se puede emplear un documento instructivo paso a paso.

Automatizadas: Se necesita de un framework automatizado para escribir los scripts de prueba.

#### ¿Qué se necesita para automatizar los unit test?

Test Runner: Es una herramienta que ejecuta los test y muestra los resultados en forma de reporte.

Por ejemplo: Mocha (https://mochajs.org/)

Assertion Library: Es una herramienta que se utiliza para validar la lógica de prueba, las condiciones y resultados esperados.

Por ejemplo: Chai (<https://www.chaijs.com/guide/>)

#### Frameworks más utilizados

Junit: herramienta de prueba de uso gratuito que se utiliza para el lenguaje de programación Java. Proporciona afirmaciones para identificar el método de prueba. Esta herramienta prueba los datos primero y luego los inserta en el fragmento de código.

NUnit: es un marco de trabajo de pruebas unitarias ampliamente utilizado para todos los lenguajes .net. Es una herramienta de código abierto y admite pruebas basadas en datos que pueden ejecutarse en paralelo.

JMockit: es una herramienta de prueba unitaria de código abierto. Es una herramienta de cobertura de código con métricas de sentencia y decisión. Permite hacer mocks de API con sintaxis de grabación y verificación. Esta herramienta ofrece cobertura de sentencia, cobertura de decisión y cobertura de datos.

EMMA: es un conjunto de herramientas de código abierto para analizar y reportar código escrito en lenguaje Java. Emma admite tipos de cobertura como método, sentencia, bloque básico. Está basado en Java, por lo que no tiene dependencias de bibliotecas externas y puede acceder al código fuente.

PHPUnit: es una herramienta de prueba unitaria para programadores PHP. Toma pequeñas porciones de código que se denominan unidades y prueba cada una de ellas por separado. La herramienta también permite a los desarrolladores usar métodos de confirmación predefinidos para afirmar que un sistema se comporta de cierta manera.

### Test unitario 3 con JavaScript

Para JavaScript vamos utilizar el framework JEST. Si queremos configurarlo, debemos instalar:

1. NodeJS
2. Un IDE (el recomendado es Visual Studio Code )
3. 3. JEST (<https://jestjs.io/>)

### Técnicas de prueba de caja blanca

Las técnicas de prueba de caja blanca, también conocidas como pruebas estructurales, se basan en la estructura interna del objeto de prueba, es decir, que está fuertemente ligado al código fuente.

Estas técnicas se pueden utilizar en todos los niveles de prueba.

Cuando se crean casos de prueba con este tipo de técnicas es aconsejable utilizar también las técnicas de caja negra como partición de equivalencia y análisis de valores límites. De este modo se conseguirán datos de prueba que maximicen la cobertura de prueba.

Las siguientes técnicas se utilizan con mayor frecuencia en el nivel de prueba de componente:

#### Prueba y cobertura de sentencia

* Ejercita las sentencias ejecutables en el código.
* Expone código que nunca se ejecuta o que se encuentra bajo condiciones imposibles.
* Cuando se logra una cobertura del 100% de sentencia, se asegura de que todas las sentencias ejecutables del código se han probado al menos una vez, pero no asegura de que se haya probado toda la lógica de decisión. Por lo tanto, la prueba de sentencia puede proporcionar menos cobertura que la prueba de decisión.

La cobertura se mide como:

#### Prueba y cobertura de decisión

* Ejercita las decisiones en el código y prueba el código que se ejecuta basado en los resultados de la decisión.
* Los casos de prueba siguen los flujos de control que se producen desde un punto de decisión.
* En el caso de un IF se necesitan dos casos de prueba como mínimo, uno para el valor VERDADERO y otro para el FALSO de la decisión.
* En el caso de un CASE se necesitan casos de prueba para todos los resultados posibles, incluido el por defecto.
* Ayuda a encontrar defectos en el código que no fueron practicados por otras pruebas ya que se deben recorrer todos los caminos de una decisión.
* Cuando se alcanza el 100% de cobertura de decisión, se ejecutan todos los resultados de decisión. Esto incluye probar el resultado verdadero y también el resultado falso, incluso cuando no hay una sentencia falsa explícita.
* Lograr una cobertura del 100% de decisión garantiza una cobertura del 100% de sentencia, pero no al revés.

La cobertura se mide como:

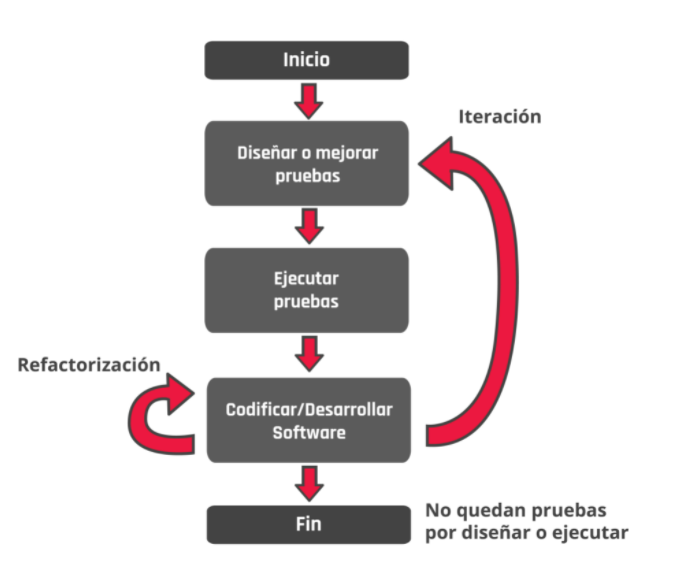
### TDD - Test Driven Development o Desarrollo guiado por pruebas

El desarrollo ágil exige comentarios periódicos para desarrollar el producto esperado. En términos simples, también se lo puede denominar como desarrollo impulsado por retroalimentación.

Existe una alta probabilidad de que los requisitos del proyecto cambien durante el ciclo del sprint de desarrollo. Para lidiar con esto y construir productos alineados con los requisitos cambiantes del cliente, los equipos necesitan retroalimentación constante para evitar distribuir software inutilizable. TDD está diseñado para ofrecer tal retroalimentación desde el principio.

El enfoque de “prueba primero” de TDD también ayuda a mitigar los cuellos de botella críticos que obstruyen la calidad y la entrega del software. Basado en la retroalimentación, la corrección de errores y la adición de nuevas funciones, el sistema evoluciona para garantizar que todo funcione según lo esperado. TDD mejora la colaboración entre los miembros del equipo tanto del desarrollo como de los equipos de control de calidad, así como con el cliente. Además, como las pruebas se crean de antemano, los equipos no necesitan perder tiempo recreando extensos scripts de prueba.

El método convencional de testing plantea tomar funciones y componentes, analizar sus casos de uso y escribir los tests cubriendo las distintas alternativas encontradas. TDD, sin embargo, propone que lo primero que se debe hacer es escribir los tests y luego el software.



La metodología nos aporta distintos beneficios que surgen, principalmente, de pensar y comprender primero el problema en su totalidad, antes de plantear la solución:

* Al redactar primero los criterios ganamos visibilidad sobre la totalidad del problema a solucionar y, recién habiendo hecho esto, procedemos a escribir código.
* Nos obliga a plantearnos cómo será el uso e interacción del código escrito al tener que pensar primero en su test.
* Nos fuerza a solucionar un problema a la vez.
* Nos permite iterar una vez que tenemos un código base funcional. Nos quita la presión de escribir un código prolijo y performante al primer intento. Primero hay que enfocarse en que funcione, luego hay tiempo para la mejora.
* Nos ayuda a no escribir más código del necesario. Al ir punto por punto y ateniéndonos a escribir el código necesario únicamente para cumplir el criterio de aceptación, nos despojamos de casos de uso rebuscados o la sobreescritura de código. Únicamente escribimos lo necesario.

TDD es una metodología muy útil, pero no es adoptada por todas las empresas o en todos los proyectos. Como consecuencia, implica dedicarle más tiempo a cada tarea, lo que a su vez se traduce en tiempo de desarrollo y costos para el empleador.

### TDD vs BDD

|  |  |  |
| --- | --- | --- |
| Desarrollo guiado por pruebas (TDD) | Vs | Desarrollo guiado por el comportamiento (BDD) |
| Es un proceso de desarrollo de software donde se desarrolla el código guiado por casos de prueba automatizados. | **Definición** | Es un proceso de desarrollo de software que permite al desarrollador concentrarse en probar el código basándose en el comportamiento esperado del software. |
| Las pruebas escritas son principalmente de nivel unitario y se centran en el código, aunque también pueden escribirse pruebas a nivel de integración o de sistema. | **Nivel de prueba** | Las pruebas escritas son principalmente de nivel de sistema e integración, aunque también se pueden utilizar para escribir pruebas unitarias.Las pruebas escritas son principalmente de nivel de sistema e integración, aunque también se pueden utilizar para escribir pruebas unitarias. |
| Ayuda a los desarrolladores a concentrarse en resultados esperados claramente definidos. | **Utilidad** | Ayuda al desarrollador o probador a colaborar con otras partes interesadas para definir pruebas precisas centradas en las necesidades del negocio. |
| Existe menor redundancia debido a que las pruebas se automatizan y se utilizan en la integración continua. | **Redundancia/Retrabajo** | Existe menor retrabajo debido a que las pruebas suelen ser más fáciles de entender para los demás miembros del equipo y los implicados. |
| Mayor calidad en el código desarrollado. | **Calidad** | Mayor calidad en software debido a que todo el equipo puede entender y colaborar en las pruebas. |
| Mayor productividad debido a que hay un menor tiempo de debugging. Menor comunicación debido a que para entender las pruebas se necesita conocer un lenguaje técnico. | **Productividad** | Mayor productividad debido a que los casos de prueba se pueden compartir con todas las partes interesadas y los frameworks utilizados generan métricas en forma automática. Mejora la confianza entre los miembros del equipo. Mayor retroalimentación con el cliente. |

Tabla 4: TDD Vs BDD

#### Procesos

###### TDD

El proceso consiste en:

1. Se añade una prueba que capture el concepto del programador sobre el funcionamiento deseado de un pequeño fragmento de código.
2. Se ejecuta la prueba, que debería fallar, ya que el código no existe.
3. Se escribe el código y se ejecuta la prueba en un bucle cerrado hasta que la prueba pase.
4. Se refactoriza el código después de que la prueba haya sido exitosa, y se vuelve a ejecutar la prueba para asegurarse de que sigue pasando contra el código refactorizado.
5. Se repite este proceso para el siguiente pequeño fragmento de código, ejecutando las pruebas anteriores, así como las pruebas añadidas.

###### BDD

El proceso consiste en:

1. Se busca un lenguaje común, llamado lenguaje natural, para unir las especificaciones técnicas y los requisitos del cliente / negocio (historias de usuario), generalmente se utiliza Gherkin.
2. Se definen los criterios de aceptación de cada historia de usuario. Pueden utilizarse marcos de desarrollo guiados por el comportamiento (frameworks como Cucumber, Jbehave, Specflow) para definir criterios de aceptación basados en el formato dado / cuando / entonces (Given / When / Then).

* Dado un contexto inicial,
* cuando se produce un evento,
* entonces, se aseguran algunos resultados.

1. Se escribe el código del software de acuerdo a los criterios de aceptación estructurados.
2. Se genera el código para los casos de prueba, es decir, se implementa el comportamiento para cada línea en lenguaje natural.
3. Se ejecutan los casos de prueba y se refactoriza.

### Mock y Stub

Mock: objetos preprogramados con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs, aunque sus diferencias son sutiles.

* En el mock podemos definir expectativas con todo lujo de detalles.
* Valida el comportamiento en la colaboración.

Stub: proporcionan respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden grabar información sobre las llamadas; tal como una pasarela de email que recuerda cuántos mensajes envió.

* El stub tan solo devuelve respuestas preprogramadas a posibles llamadas.
* Simula respuestas a consultas.

### Cobertura de pruebas con JEST

El test coverage es un valor utilizado como indicador para obtener visibilidad sobre la “robustez” en un proyecto, pero no es garantía de infalibilidad. Recordemos que todo depende de qué estamos testeando y cómo. Es posible tener un 100% de cobertura, pero que nuestro código tenga bugs si lo estamos testeando mal.

¿Qué porcentaje de test coverage debemos tener en un proyecto? Es lógico pensar que se debería apuntar a un 100% de cobertura del código, pero en la realidad no suele suceder de esta forma. ¿Por qué?

Cada proyecto suele tener partes de mayor y menor importancia. Garantizar un 100% de cobertura de código puede implicar dedicarle tiempo a testear partes del código repetitivas o no vitales y, posiblemente, descuidar otras que necesiten más atención. Los equipos de desarrollo suelen enfocar sus esfuerzos en las partes centrales de un proyecto y garantizar que ahí haya un porcentaje de cobertura mayor o total.

A su vez, hay varias maneras y convenciones de manejar los porcentajes de cobertura de código. Si un proyecto ya comenzó sin tests y se decide agregarlos durante el transcurso de este, se suelen poner objetivos de cobertura de código e incrementarlos entre cada sprint. Se puede comenzar con buscar alcanzar un 50% de cobertura y luego subir un 5% entre cada sprint, por ejemplo.

Otra alternativa es no bajar de cierto porcentaje. Supongamos que estamos en un proyecto en el cual ya hay un 85% de cobertura de código y el equipo se propone no bajar de un 75%, como mínimo. En un caso así, es posible elegir cuándo y dónde priorizar la resolución de una feature para poder llegar a los objetivos del sprint frente a escribir todos los tests necesarios. Muchas veces se suele diferir la escritura de estos para poder cumplir con los objetivos generales del proyecto a tiempo.

Como toda convención, hay muchísimas variantes y dependerá de la empresa, el equipo y el proyecto en el que trabajes la metodología que se implemente y cómo la lleven a cabo.

En JEST el reporte de cobertura de prueba viene integrado con el framework. Simplemente debemos escribir el comando npm run test:coverageen la terminal para ejecutar los tests. Pero antes es necesario realizar algunas configuraciones en nuestro proyecto.

Veamos cómo es el paso a paso para obtener el reporte de cobertura y cómo podemos mejorar la cobertura de:

* Sentencia (Statements)
* Decisión (Branches)
* Funciones (Functions)
* Líneas (Lines)

Paso 1: Configuración de JEST para poder ejecutar el reporte de cobertura

Se debe agregar el comando test:coverage con el valor jest --coverage dentro del paquete de configuración del proyecto, archivo package.json. El mismo se agrega en el nodo “scripts”.

Paso 2: Ejecutar el reporte de cobertura

Ejecutar el reporte de cobertura desde la terminal del proyecto con el comando npm run test:coverage

En esta pantalla se detallan los archivos y una tabla de porcentajes de cobertura que responden a cada parte del código.

Este reporte generado por Jest tiene un beneficio extra: automáticamente nos genera un archivo HTML que amplía aún más estos resultados, mostrándonos inclusive que líneas son las que no se encuentran testeadas. Este reporte se agrega al proyecto en la carpeta \_\_coverage\_\_ donde se encuentra el archivo index.html que nos permite acceder al reporte desde un navegador web.

Paso 3: Revisar el reporte de cobertura

En el mismo figura todos los archivos de código que poseen pruebas relacionadas. Para ver el detalle de la cobertura se puede ingresar haciendo clic en el nombre del archivo.

En el detalle se puede ver exactamente qué líneas de código tienen cobertura y analizar si es necesario agregar más casos de prueba unitarios para alcanzar una mayor cobertura.

Paso 4: Necesitar mayor cobertura

En el caso de que sea necesario alcanzar una mayor cobertura, se pueden agregar más casos de prueba y luego ejecutar nuevamente el comando npm run test:coverage desde la terminal para regenerar el reporte.

En la pantalla siguiente se puede ver como se mejoró la cobertura agregando los casos de prueba para la función validateInput.

Finalmente se puede volver a analizar el reporte en detalle desde el navegador web y decidir si es necesario mejorar la cobertura.

Clase 17: Introducción a API Testing

# Módulo 8: BE testing

## Introducción API Testing

### ¿Qué es el testing de back end?

Una aplicación web típica tiene tres capas: interfaz (UI), lógica empresarial y una base de datos. Probar la interfaz, es decir el frontend, implica validar aquellas partes de la aplicación que son visibles para los usuarios finales, por ejemplo: formularios, menús, navegaciones, etc. Por otro lado, las pruebas de back end tratan con todos esos elementos que los usuarios no pueden ver (lógica empresarial y base de datos).

Entonces el testing back end nos garantiza que los datos contenidos en la base de datos de una aplicación y su estructura satisfagan los requisitos del proyecto.

### Introducción a APIs

Una APPI (application programming interface) es básicamente una url que no devuelve una página web como estamos acostumbrados a ver si no que lo que devuelves información para qué otro sistema lo consuma.

Las APIs son desarrolladas para que dos sistemas puedan comunicarse ya que si un sistema se aísla no tiene futuro. Como la información es para otro éste debe saber utilizarla es por esto que las APIs suelen tener su propia documentación.

Hay APIs que son públicas, otras privadas o también las hay semipúblicas. Las públicas son APIs para las cuales no necesitamos nada para consumirlas, tal vez necesitemos registrarnos, pero son gratuitas y podemos consultar su información. Por otro lado, tenemos las APIs semipúblicas donde se establece un límite de información y si requiero más debo pagar. Por último, existen las APIs privadas, en este caso los endpoints no están disponibles para nuestro consumo, son solamente para productos de la organización que las crea.

#### Endpoint

Es un punto de conexión donde necesitamos apuntar para obtener la información que queremos. Es decir, son las URL que debemos utilizar para obtener la información de un servidor a través de una API.

### Testing de APIs

Hablando en un alto nivel, nuestro trabajo es entender cómo funciona la API, armar una buena combinación de parámetros de entrada, ejecutar las pruebas contra la API, verificar el resultado y reportar cualquier desviación en la funcionalidad esperada. Estas pruebas consisten en hacer peticiones HTTP (get, post, put y delete) y luego verificar la respuesta.

### HTTP y sus métodos

El protocolo de transferencia de hipertexto (hypertext transfer protocol o HTTP) es un sencillo protocolo cliente-servidor que articula los intercambios de información entre un servidor y una aplicación que consume estos servicios. Esta comunicación se logra gracias a los métodos HTTP, los cuales nos permiten enviar y recibir información.

A la hora de comunicarnos no importa si el punto es el lenguaje o el canal o cualquier otro factor, tenemos protocolos de comunicación que definen las reglas y códigos de comunicación necesarios para entendernos. Cuando nos encontramos dentro de un esquema web hay que tener en cuenta que el mismo se rige, entre varias cosas, a través del protocolo HTPP. Este protocolo define una serie de lineamientos que se deben cumplir para procesar la información.

HTTP es un protocolo que gestiona las transacciones entre dos entidades: el cliente y el servidor. Cada vez que se hace un pedido de un cliente a un servidor este pedido viene acompañado de un método o verbo con los que trabaja http. Un método es una función que implementa una serie de procesos lógicos que definen lo que sucederá cuando se accede un recurso. Si bien dentro del protocolo http existen diversos métodos los más utilizados son GET, POST, PUT, DELETE.

##### GET

Es un método a través del cual vas a poder solicitar cosas al servidor, conceptualmente las peticiones hechas con GET sólo deben recuperar datos, es decir, no modificar información que se encuentra el servidor. Cuando accedes a una página escribiendo una URL en el navegador la petición está viajando por GET. Es importante tener presente que solamente las rutas que usen el método podrán ser accedidas desde la URL del navegador o desde un enlace. Los otros métodos necesitarán ser procesados generalmente desde un formulario.

##### POST

Es un método a través del cual puedo enviar datos al servidor. En general las peticiones hechas con POST servirán para crear registro dentro de una aplicación. Cuando nos registramos y escribimos los datos que nos solicitan, la petición va a viajar por POST.

Diferencias Técnicas

POST es un método más seguro para enviar información del cliente al servidor si lo comparamos con el GET. Cuando trabajamos con pedidos GET toda la información que enviemos será visible en la URL el navegador, por el contrario, cuando el pedido es por POST esta información está oculta. Además, los pedidos por POST no pueden ser cacheados ni estar en favoritos ni en el historial del navegador.

Los carteles de “¡Está seguro que quiere reenviar esta información?” que aparecen cuando refrescamos la página son consecuencia de los pedidos por POST.

##### PUT

Se usa para reemplazar información actual en un registro ya presente en el servidor. A diferencia de POST no creo un nuevo registro, sino que edito uno ya existente.

##### DELETE

Borra un recurso presente en el servidor.

Para estos dos últimos casos vamos a necesitar información que identifique ese registro que estamos por editar o eliminar y para esto se suele utilizar normalmente el ID del registro que ya existía. Un dato a tener en cuenta es que no todos los navegadores implementan PUT y DELETE. Con lo cual existen frameworks con algunas prestaciones para simularlos.

Clase 19: API Testing

## API Testing

### Método GET

Utilizaremos una solicitud GET para recuperar información de una URL específica y analizar la información obtenida a partir de los test.

#### Pasos

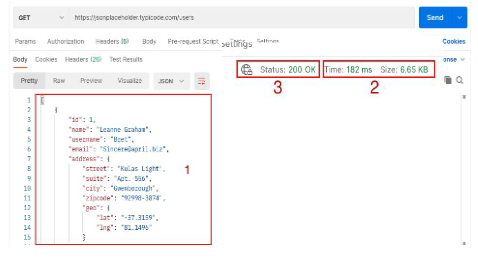
1. Primero, debemos crear una nueva solicitud en Postman. Para realizarlo, se debe hacer clic en la pestaña ‘’New’’
2. El siguiente paso es crear la solicitud. ¿Cómo?
   1. Configura su solicitud HTTP en GET.
   2. Ingresa el enlace en la URL de la solicitud (https://jsonplaceholder.typicode.com/users)
   3. Haz clic en ENVIAR para mandar la solicitud al servidor que aloja la URL
3. Cuando aparezca el mensaje 200 OK significa que se la solicitud se realizó correctamente.

#### Resultados

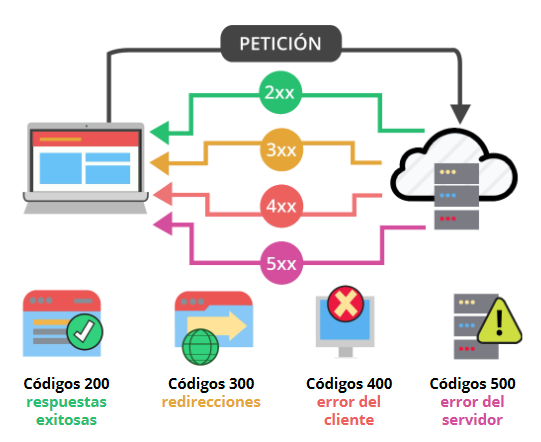
Response (1): es la información en plano devuelta por el servidor. Con esto podemos dar una revisión temprana de los datos de la aplicación.

Tiempo y tamaño de la respuesta (2): con estos datos podemos ver si el sistema está cumpliendo uno de los requisitos no funcionales, tal como el rendimiento.

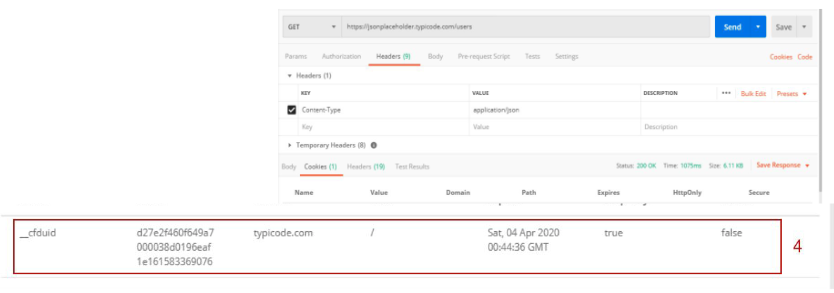
Código de respuesta (3): cuando solicitas información al servidor, este puede contestar distintos códigos de estado que te informan qué pasó con tu solicitud. El código 200 nos indica que la solicitud se realizó con éxito.



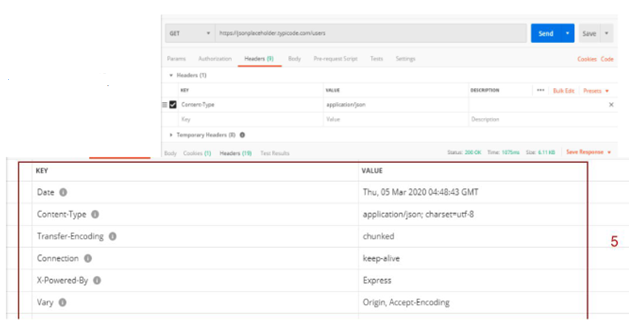
Un API nos devuelve diferentes códigos de respuesta que nos informan qué pasó con la petición. Estas respuestas se agrupan en cuatro clases.



Cookies: nos permiten ver la información relacionada con la sesión.



Headers: información sobre la solicitud procesada.



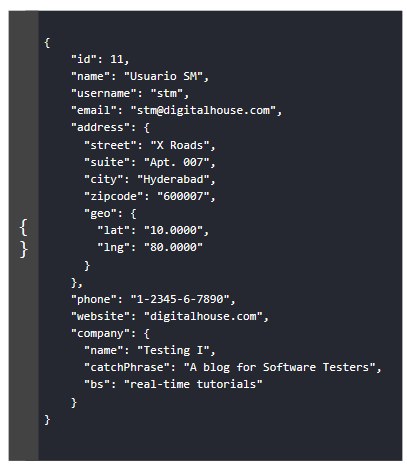
### Método POST

Cuando necesitamos agregar datos a nuestra aplicación utilizamos el método POST para enviar estos datos. A través de esta solicitud enviamos los datos y el API nos devuelve una respuesta que valida que la creación sea exitosa.

#### Pasos

1. Al igual que con el método GET, se debe crear una nueva solicitud en Postman. Entonces, debemos hacer clic en la pestaña ‘’New”.
2. El siguiente paso, es crear la solicitud. ¿Cómo?
   1. Configurar su solicitud HTTP en POST.
   2. Ingresar el enlace en la URL de la solicitud (<https://jsonplaceholder.typicode.com/users>).
   3. Los datos para una petición POST no se los pasa por la url porque no viajan seguros: se pasan por el BODY. Lo podemos enviar de diferentes formas:

* Raw: se envía la información como una cadena tipo texto, a través de un archivo tipo JSON.
* x-www-form-unlencoded: se envían los datos como si fuera un formulario



1. En este ejemplo enviaremos los datos en formato Raw. Para ello haz clic en el cuerpo de la solicitud y selecciona la opción “raw” (a), luego “Json” (b). Finalmente copia y pega el ejemplo brindado en la diapositiva anterior en el body (c)



1. Haz clic en ENVIAR para mandar la solicitud al servidor que aloja la URL.
2. Si aparece el mensaje 201 CREATED, significa que se la solicitud se realizó correctamente.

#### Resultados

Response: es la información en plano devuelta por el servidor, la cual nos sirve para validar si la creación fue exitosa. Generalmente este método devuelve los datos del usuario creado o un mensaje de creación exitosa.

Tiempo y tamaño de la respuesta: con estos datos podemos ver si el sistema está cumpliendo uno de los requisitos no funcionales, tal como el rendimiento.

Código de respuesta: cuando solicitas información al servidor, este puede contestar distintos códigos de estado que te informan qué pasó con tu solicitud. El código de respuesta 201 nos indica que la creación fue exitosa.

### API’s Testing - Automatización de pruebas

#### ¿Qué son las pruebas automatizadas?

Las pruebas se automatizan mediante la creación de conjuntos de pruebas que se pueden ejecutar una y otra vez (scripts), y no requieren ninguna intervención manual. Postman se puede utilizar para automatizar muchos tipos de pruebas, incluidas las unitarias, las funcionales, de integración, de extremo a extremo, de regresión, las simuladas, etc.

Las pruebas automatizadas evitan errores humanos y agilizan las pruebas.

#### ¿Por qué automatizar una prueba?

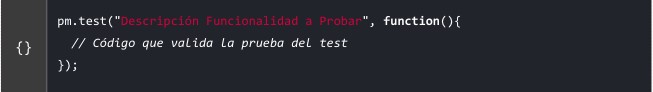
A medida que los programas crecen, también aumenta el riesgo de rotura. Se pueden crear programas más robustos y resistentes a errores aumentando la cobertura y la frecuencia de las pruebas. Postman permite reutilizar sus conjuntos de pruebas para simplificar el trabajo de forma más efectiva y productiva.

### API Testing- Creación de pruebas (JS)

#### Pasos

1. Comencemos con algunos conceptos a tener en cuenta:

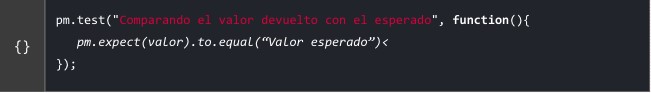
* Para codificar los test con [Postman](https://www.arquitectoit.com/postman/que-es-postman/) debemos de conocer un poco el API que nos ofrecen. Cada uno de los test es ejecutado con el objeto pm y en concreto con el método .test(). Así, para cada uno, tendremos la siguiente estructura.



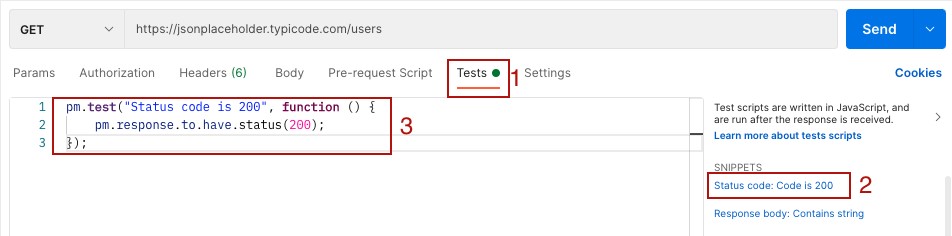
* Para poder acceder al contenido de la respuesta de las invocaciones tenemos el objeto pm.response y su método .json() que nos permitirán acceder a los elementos de la respuesta en JSON.



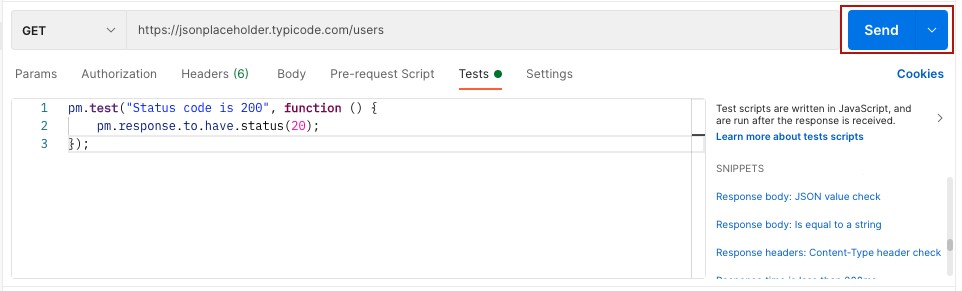
* Otro método importante es el que nos permite realizar una comprobación de contenido, este es pm.expect.



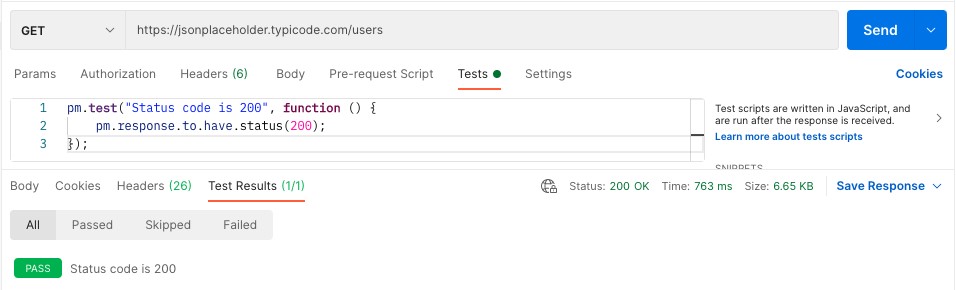
1. Teniendo en cuenta los conceptos ya definidos, veremos dos de las pruebas más utilizadas en el Testing de APIs. Postman nos brinda una serie de fragmentos por defecto que nos guía a la hora de construir nuestras pruebas:
2. Para comenzar, vamos a la solicitud GET que creamos anteriormente y seleccionamos la pestaña Pruebas (1). En esta sección escribiremos nuestro set de pruebas relacionados con esa API. En la subsección de fragmentos, haremos clic en "Código de estado: el código es 200" (2) para generar una de las pruebas por defecto. La secuencia de comandos se completará automáticamente (3).



1. Al hacer clic en ENVIAR se mostrará el resultado del test.

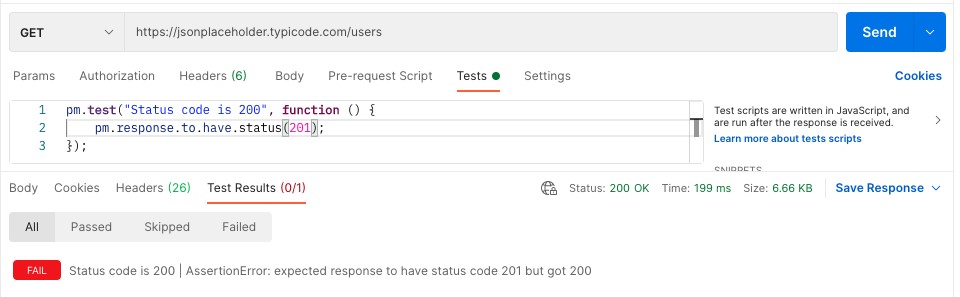


1. Con esta prueba estamos validando que el código de respuesta de la API sea 200. Si esto es correcto el test devolverá PASS: eso significa que el servicio está respondiendo según lo esperado.

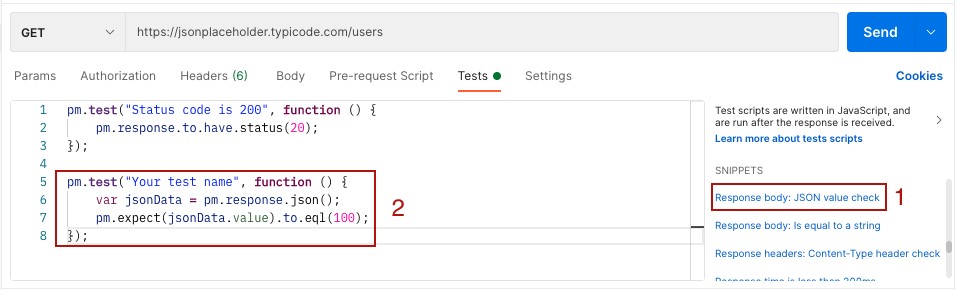


1. Si el servicio falla nos mostrará el estado FAIL, y el código de error relacionado con este estado.

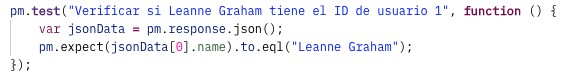
En esta prueba estamos reutilizando fragmentos de código que nos brinda Postman para validar si la petición se realizó correctamente. Podemos editar esta consulta a nuestro gusto utilizando código javascript.



1. Agreguemos otra de las pruebas más usadas. En esta compararemos el resultado esperado con el resultado real.
2. Para ello, en la subsección de fragmentos, haremos clic en "Cuerpo de respuesta: Verificación del valor del JSON" (1). La secuencia de comandos se completará automáticamente (2).



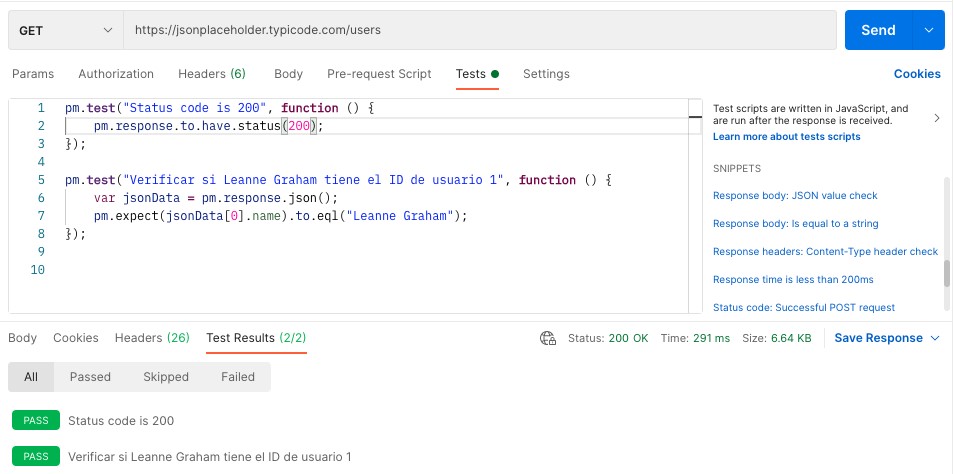
1. Podemos cambiar el nombre de la prueba por defecto por el que más nos guste. En este caso lo reemplazamos por “Verificar si Leanne Graham tiene el ID de usuario 1”, dado que este es el primero usuario de la lista devuelta por el API. También debemos actualizar el cuerpo de la función reemplazando jsonData.value con jsonData[0].name; así obtendremos el primer elemento de la lista.



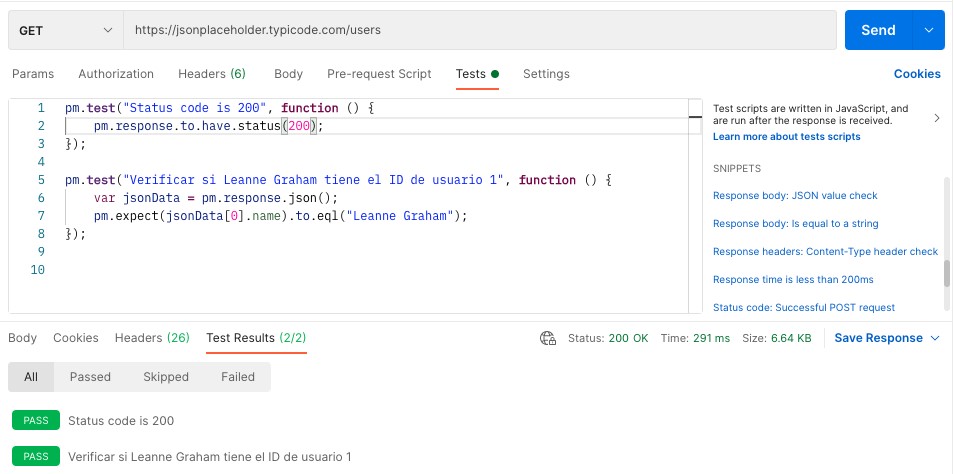
1. Al hacer clic en ENVIAR se mostrará el resultado.



1. Se observa que nuestro test nos devolvió un estado “PASS”. De esta manera validamos que el contenido de la response es el esperado. Así, podremos ir validando diferentes datos y viendo si nuestra petición nos devuelve los datos deseados.



1. Por último, se observa que al momento de enviar la petición se ejecutan todos los test relacionados a esta. De esta manera, podemos crear un set de test vinculados a cada petición y verificar rápidamente su estado.



## Colecciones y variables de entorno con Postman

### Colecciones

Las colecciones son un grupo de peticiones guardadas que pueden organizarse en carpetas. Esto nos permite agrupar y administrar nuestras peticiones de manera más eficiente.

##### Pasos para crear una colección

1. Primero, se debe hacer clic en el botón ‘’Crear nueva Colección’’ dentro de la pestaña colecciones.
2. El siguiente paso es completar los datos de la nueva colección.
   1. Ingresa el nombre de la colección.
   2. Ingresa una descripción para la misma (opcional).
   3. Haz clic en CREAR para crear la nueva colección.
3. Puedes agregar cualquier número de peticiones a una colección. Solo debes arrastrar la petición a la carpeta de la colección

### Variables de entorno

Solemos utilizar la misma solicitud varias veces con datos diferentes. Postman nos permite parametrizar estos datos y guardarlos en forma de archivo o en variables de entorno.

Una variable de entorno se guarda en el entorno de trabajo. Estas se pueden crear de manera estática o dinámica. Podríamos tener diferentes entornos para Dev, QA y Producción, con sus respectivas variables.

##### Pasos para crear una variable de entorno

1. Primero, se debe hacer clic en el botón del ojo para crear nuestras variables de entorno.
2. El siguiente paso es completar los datos del entorno y de las variables. ¿Cómo?
   1. Ingresa el nombre del entorno (DEV, QA o Producción).
   2. Ingresa las variables de ese entorno. Para ello hay que completar el nombre de la variable y su valor.
   3. Finalmente haz clic en el botón Guardar para efectuar los cambios.
3. Por último, se crean mediante el uso de llaves dobles con el nombre de la variable a utilizar.

### Runner para correr colecciones

El runner nos permite ejecutar un conjunto de test de diferentes colecciones al mismo tiempo, otorgando un informe de resultados.

##### Pasos para ejecutar el runner

1. Primero, se debe hacer clic en el botón Runner.
2. El siguiente paso es completar los datos de la ejecución ¿Cómo?
   1. Seleccionamos las colecciones y peticiones en las que deseamos ejecutar sus test.
   2. Seleccionamos el entorno en cual deseamos correr nuestros test. De esta manera se utilizarán las variables relacionadas con ese entorno.
   3. Indicamos la cantidad de veces que vamos a correr los test.
   4. Puedes configurar el tiempo de demora entre prueba.
   5. Puedes seleccionar un archivo para guardar sus pruebas y resultados.
   6. Se pueden guardar las cookies para utilizarlas en otros test.
   7. Haz click en el botón Empezar Ejecución.
3. Por último, podemos ver el resultado de nuestras ejecuciones.

Clase 20: Fundamentos de Automatización de la prueba

# Módulo 9: Introducción a la Automatización

## Pruebas automatizadas

Es el proceso de ejecutar varias pruebas una y otra vez sin intervención humana utilizando una herramienta de automatización. Las pruebas manuales y las pruebas automatizadas no son excluyentes: se complementan.

### Tipos de pruebas automatizadas

Se pueden automatizar los siguientes tipos de pruebas:

* Pruebas unitarias.
* Pruebas de API.
* Pruebas de interfaz gráfica.

Otras pruebas que también pueden automatizarse son las de rendimiento, de regresión, de integración, de seguridad, pruebas de compatibilidad en diferentes navegadores, casos repetitivos.

### Objetivos de automatización

* Mejorar la eficiencia de la prueba
* Aportar una cobertura de funciones más amplia
* Reducir el costo total de las pruebas
* Acortar el período de ejecución de las pruebas
* Realizar pruebas que no se pueden hacer manualmente

### Ventajas y Desventajas

|  |  |
| --- | --- |
| VENTAJAS | DESVENTAJAS |
| * Pruebas complejas. * Ahorro de tiempo en la ejecución de pruebas. * Las pruebas están menos sujetas a errores del tester. * Se pueden rehusar los scripts con pruebas automatizadas | * El equipo necesita tener conocimientos técnicos para poder implementar los scripts de test. * Requiere un mantenimiento continuo. * No podemos automatizar aspectos como el nivel de usabilidad. * Requiere tecnologías adicionales. |

Tabla 5: Ventajas y Desventajas de la Automatización de la prueba

## Patrones de Diseño

Los patrones de diseño son soluciones probadas y documentadas a problemas comunes de desarrollo de software. También son usados en la automatización de software.

### Patrones de diseño en automatización

#### Screenplay

* Este patrón tiene un enfoque de desarrollo encaminado por el comportamiento Behaviour Driven Development (BDD). Es una estrategia de desarrollo que se enfoca en prevenir defectos en lugar de encontrarlos en un ambiente controlado.
* Presenta un alto desacoplamiento de la interfaz de usuario.
* Propone trabajar en términos de tareas y no de interfaz de usuario.
* Cumple con los principios SOLID.

#### Page Object

* Es un patrón de diseño que representa los componentes web o página web en una clase.
* Se utiliza en las pruebas automatizadas para evitar código duplicado y mejorar el mantenimiento de las mismas.
* No cumple con los principios SOLID.

### ¿Qué son los principios SOLID?

SOLID es un acrónimo introducido por Robert C.Martin para definir los cinco principios básicos de la programación orientada a objetos: Single responsibility, Open-closed, Liskov substitution, Interface segregation y Dependency inversion. SOLID tiene bastante relación con los patrones de diseño.

Los principios SOLID son guías, buenas prácticas, que pueden ser aplicadas en el desarrollo de software para eliminar malos diseños. Provocan que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Es decir, que pueden ayudar a escribir un mejor código: más limpio, mantenible y escalable.

Algunos de los objetivos a tener en cuenta estos principios para código son:

* Crear un software eficaz que cumpla con su cometido y que sea robusto y estable.
* Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
* Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

En definitiva, desarrollar un software de calidad.

Principio de responsabilidad única

Como su propio nombre lo indica establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término *decoupled* en inglés). Si, por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra.

Principio abierto/cerrado

Establece que las entidades*software* (clases, módulos y funciones) deberían estar abiertas para su extensión, pero cerradas para su modificación.

Principio de substitución de Liskov

Declara que una subclase debe ser sustituible por su superclase. Si al hacer esto el programa falla estaremos violando este principio. Cumpliendo con este se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

Principio de segregación de interfaz

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este no usa, pero que otros clientes sí lo hacen, este estará siendo afectado por los cambios que fuercen los demás en dicha interfaz.

Principio de inversión de dependencias

Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

* Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
* Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.

En algún momento nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase debemos usar inyección de dependencias: nos permitirá controlar las funcionalidades desde un sitio concreto en vez de tenerlas esparcidas por todo el programa.

## Page Object Model

Page Object Model, también llamado POM, es la implementación del patrón de diseño Page Object utilizado en la automatización de pruebas. Tiene como objetivo mejorar el mantenimiento de las pruebas y reducir la duplicación de código.

El concepto básico en el que se basa el Page Object Model es el de representar cada una de las pantallas que componen al sitio web o la aplicación que nos interesa probar como una serie de objetos que encapsulan las características (localizadores) y funcionalidades representadas en la página. De esta manera, nos permite consolidar el código para interactuar con los elementos de una página en cada uno de los PageObjects.

Las páginas web se representan como clases y los diversos elementos de la página se definen como variables en la clase. Todas las interacciones de usuario posibles se pueden implementar como métodos en la clase.

Con lo cual, cualquier cambio que se produzca en la UI únicamente afectará al PageObject en cuestión, no a los test ya implementados.

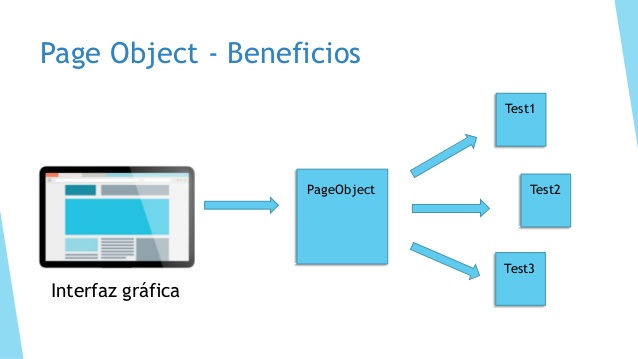
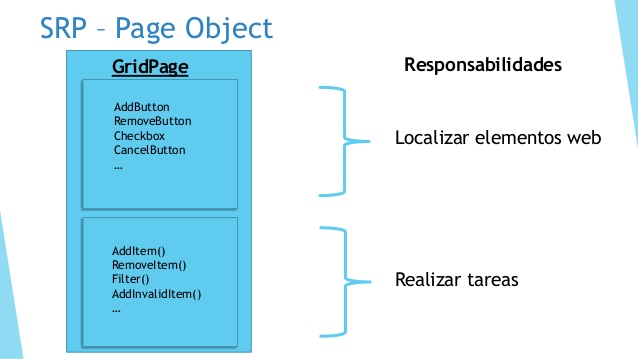


Ilustración 3: Beneficios del Page Object

Para que el Page Object Model cumpla con los principios SOLID debemos refactorizar, en este ejemplo de abajo, la Clase GridPage y separar las responsabilidades de localizar los elementos de las funcionalidades. Con lo cual una opción es realizar una clase con los localizadores “GridPage” y una clase por cada una de las funcionalidades, como se observa en la imagen:



## Introducción a Selenium

Selenium fue desarrollado por Jason Huggins en 2004. Es un framework destinado a la automatización web que consiste en desarrollar scripts que, mediante algún lenguaje de codificación determinado, permite ejecutar un flujo de navegación fijo. De este modo, garantiza que el comportamiento de dicho flujo se conserve a lo largo de la vida de la página web.

Es una herramienta de código abierto para la automatización de pruebas de navegadores web. Proporciona la posibilidad de grabar y/o reproducir, editar y depurar casos de pruebas que permitirán ejecutarlas repetidamente cuando sea necesario. Selenium ofrece tres productos con distintos propósitos:

Selenium Grid: Permite diseñar pruebas automatizadas para aplicaciones web en diversas plataformas. Asimismo, posibilita la ejecución de pruebas en diversos servidores en paralelo. Es por esto por lo que reduce el tiempo de ejecución y el costo, debido a la ejecución de las pruebas en varios navegadores y en diversos sistemas operativos. Selenium Grid cuenta con dos componentes: Selenium Hub y Remote Control.

Selenium WebDriver: Es una herramienta que permite automatizar pruebas UI (User Interface) o interfaz de usuario de aplicaciones web. Algunos de los lenguajes que son soportados son Java, C#, Python, Ruby, PHP, JavaScript. Proporciona APIs orientadas a objetos en una variedad de idiomas. Esto nos permite tener un mayor control sobre la aplicación de las prácticas de desarrollo de software estándar. Es útil para poder simular la manera en que los usuarios reales interactúan con alguna aplicación web.

Selenium IDE (Integrated Development Environment - Entorno de Desarrollo Integrado): este componente es una herramienta de automatización que nos permite grabar, editar y depurar pruebas. También se lo conoce como Selenium Recorder.

### Selenium IDE

Se trata de un entorno de pruebas de software para aplicaciones basadas en la web que permite realizar tareas de Record&Play de flujos de pruebas. Los flujos grabados quedan contenidos en un script que se puede editar y parametrizar para adaptarse a los diferentes casos, y lo que es más importante, su ejecución se puede repetir tantas veces como se quiera.

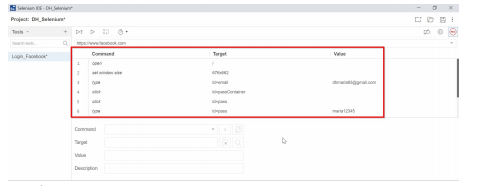
Su principal objetivo es automatizar pruebas funcionales repetitivas para luego facilitar el trabajo del tester, como también pruebas de regresión. Permite referenciar objetos del DOM en base al ID, CSS, nombre o a través de XPath. Asimismo, las acciones pueden ser ejecutadas paso a paso.

#### Ejemplo Record & Play

En esta oportunidad vamos a crear/grabar el caso de prueba de login de Facebook. Para ello debemos realizar los siguientes pasos:

1. Presionar del navegador, la opción.
2. Seleccionar la opción “Record a new test in a new project”.
3. Ingresar nombre del proyecto.
4. Ingresar URL de la página web (siempre con el https://) y presionar la opción “Start Recording” para iniciar la grabación de la prueba.
5. Se abrirá el navegador con la página web indicada. Para nuestro ejemplo, posicionarse en el campo “Correo electrónico” e ingresar un mail. Luego, en el campo contraseña, ingresar una, y por último, hacer clic en “Iniciar sesión”. Por último, presionar la opción STOP de Selenium IDE e ingresar el nombre del caso de prueba.

##### Resultados del test

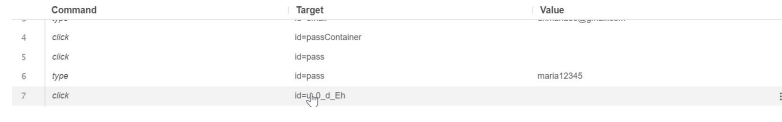


En esta grilla se visualizan:

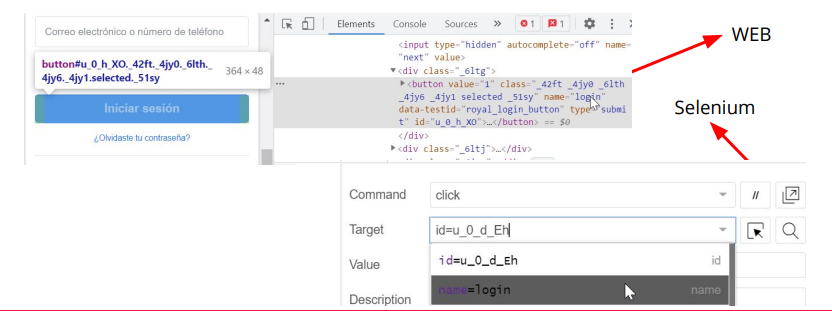
* Comandos, que serían las acciones realizadas en la grabación.
* Target: Son los localizadores de elementos web en la página web. Hay diferentes formas de ubicar elementos: ID, ClassName, Name, TagName, LinkText, PartialLinkText, Xpath, CSS Selector, DOM.
* Value: Este campo es un valor simple, almacenado en ese elemento web grabado durante la grabación. También se puede establecer un nuevo valor para pruebas siguientes

Podemos observar que se localizó el botón de inicio de sesión con id. Al reproducir este caso de prueba, una vez que llega a ejecutarlo entra en un lapso sin dar respuesta. Por lo tanto, se asume que se localizó de manera incorrecta el elemento, en este caso, el botón de inicio de sesión.

Este caso de componentes de páginas web que tienen un id dinámico se presenta con frecuencia. Para resolver esta inconsistencia se debe buscar otra forma de ubicar el componente.



Se debe inspeccionar el elemento y validar, si es posible, la localización por otro parámetro. En este caso, se puede localizar por name. Entonces, en Selenium se debe modificar el target a name:



Clase 22: Automatización de la prueba

## Selenium WebDriver

Selenium es uno de los frameworks más importantes con los que se generan pruebas automatizadas.

Es un conjunto de herramientas para la automatización de navegadores web que utiliza las mejores técnicas disponibles para controlar las instancias de los navegadores y emular la interacción del usuario con el navegador.

Permite a los usuarios simular interacciones realizadas por los usuarios finales como insertar texto en los campos, seleccionar valores de menús desplegables y casillas de verificación, y mucho más.

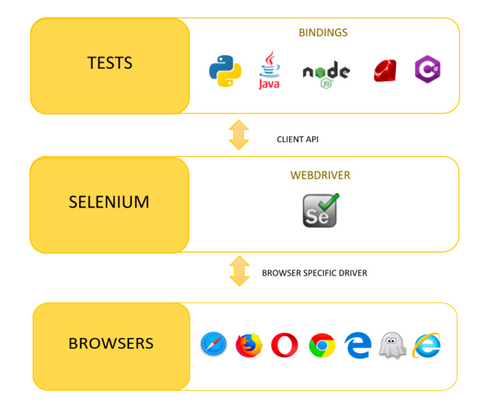
WebDriver utiliza las API de automatización del navegador proporcionadas por los desarrolladores de los navegadores para controlarlo y ejecutar pruebas. Esto es como si un usuario real estuviera manipulándolo.

Selenium es compatible con muchos lenguajes de programación como Java, C #, Python, etc., y con múltiples navegadores como Google Chrome, Firefox, Internet Explorer, etc.

La arquitectura de Selenium Webdriver se enmarca en tres componentes principales:

* Los tests a ejecutarse (escritos con un lenguaje a elección entre varios como Java, C#, Node JS, etc); “Client API”.
* Un servidor standalone en donde se ejecutarán los casos de prueba.
* Un browser driver que conectará los scripts generados con la client.
* API con el browser a través del Selenium Server.

Gráficamente puede verse así:



### Instalación de Selenium WebDriver

La configuración de Selenium es bastante diferente de la configuración de otras herramientas comerciales. Para usar Selenium en tu proyecto de automatización necesitas instalar las librerías de tu lenguaje de preferencia. Además, necesitarás los binarios de WebDriver para los navegadores en los que deseas automatizar y ejecutar pruebas.

* + 1. Instalando las librerías de Selenium: Primero debes instalar las librerías de Selenium para tu proyecto de automatización. El proceso de instalación depende del lenguaje que elijas usar. En este caso te pediremos que instales las librerías de JavaScript.

Para JS en la consola escribimos: npm install selenium-webdriver

* + 1. Instalando los binarios de WebDriver: Para ejecutar tu proyecto y controlar el navegador debes tener instalados los binarios de WebDriver específicos para el navegador.

### Creación del primer script de prueba: Login de Facebook

Una vez instalados todos los componentes, vamos a crear nuestro primer script de prueba. Nuestro caso de prueba es realizar el login en Facebook con nuestro mail y contraseña.

Para esto instanciamos al controlador, iniciamos una sesión del explorador que le pasamos como parámetro. Se localizarán los campos necesarios para la prueba, en este caso el campo mail, contraseña y el botón de inicio de sesión y, por último, terminar la sesión del navegador.



Clase 25: Full stack tester - Frameworks

# Módulo 10: Full stack y DevOps tester

## Full Stack Tester

Generalmente relacionamos el término Full Stack al rol del desarrollador. Full Stack es específico para aplicaciones basadas en web y se refiere a todo el sistema. Dado que las aplicaciones web se crean utilizando una arquitectura en capas, el desarrollo Full Stack implica hacer partes tanto del lado del cliente como del lado del servidor, es decir, desde el Front End hasta el Back End y la conexión entre ellos.

Los roles de desarrollo generalmente se dividen en Front End y Back End, aunque en algunos proyectos se requiere un desarrollador Full Stack. A partir de este modelo, la pregunta es si las personas dedicadas a la prueba de sistemas (testers) deberían estar igualmente a cargo de las pruebas Full Stack.



Existe la idea errónea de que cualquiera puede ser un tester, ya que cualquiera puede navegar a través de una aplicación, hacer clic en los botones y encontrar posibles errores. Esto es erróneo, ya que realizar pruebas implica probar conceptos y metodologías y tener una forma crítica de ver las cosas. Actualmente los testers deben estar familiarizados con la arquitectura de la aplicación. Tener una comprensión profunda de cómo funcionan las cosas puede mejorar la estrategia de prueba: saber qué, dónde y cómo probar diferentes estructuras es esencial para mejorar la calidad general.

Tener acceso a la base de datos ayudará al tester a ver si los datos del frontend se guardan adecuadamente y le ayudará a elegir los datos de prueba correctos para probar si se necesita información específica.

Respecto a las pruebas de API, es esencial que un evaluador inspeccione una página web para ver las solicitudes enviadas, las respuestas o los datos transmitidos a través de llamadas API en formato JSON. Es crucial verificar si los datos de la API coinciden con las entradas de la base de datos y los datos de la interfaz de usuario.

Por lo tanto, un sólido conocimiento de las reglas de negocio combinado con habilidades técnicas mínimas y las herramientas adecuadas son los requisitos necesarios para un excelente Full Stack Tester.

Un Full Stack Tester no lo sabe todo, sino que comprende los mecanismos y puede encontrar las mejores estrategias de prueba cuando se necesita realizar las pruebas por toda la aplicación y garantizar que se entrega un producto de calidad.

## Frameworks

Un framework define un conjunto de reglas/pautas o mejores prácticas que podemos seguir de manera sistemática para lograr los resultados deseados.

Un **framework** **de** **automatización** de pruebas es un **conjunto de pautas** como estándares de codificación, manejo de datos de prueba, tratamiento de repositorios de objetos, etc., que cuando se siguen durante la secuencia de comandos de automatización producen resultados beneficiosos. Estos pueden ser una mayor reutilización de código, mayor portabilidad, menor costo de mantenimiento de secuencias de comandos, etc.

Se trata de pautas y no reglas en el sentido estricto de la palabra ya que no son obligatorios. Se puede escribir un script sin seguir las pautas, pero se perderán las ventajas de tener un framework.

### Tipos de Frameworks

#### Framework de secuencias de comandos lineal

Conocido como “grabación y reproducción” es el más simple de todos los frameworks de automatización de pruebas. En este framework el tester registra manualmente cada paso (navegación y entradas del usuario) e inserta puntos de control (pasos de validación) en la primera iteración. Luego, reproduce el script grabado en las iteraciones siguientes.

Ejemplos: Selenium GRID, UFT, Test Complete, Sahi.

|  |  |
| --- | --- |
| Ventajas | Desventajas |
| La forma más rápida de generar un script. | Poca reutilización de guiones. |
| No se requiere experiencia en automatización. | Los datos de prueba están codificados en el script. |
| La forma más sencilla de conocer las funciones de la herramienta de prueba. | Pesadilla de mantenimiento. |

Tabla 6: Framework de secuencias de comandos lineal

#### Framework de prueba modular

En el framework de prueba modular, los probadores crean módulos de scripts de prueba dividiendo la aplicación completa bajo prueba en pruebas independientes más pequeñas.

En palabras simples, los evaluadores dividen la aplicación en varios módulos y crean scripts de prueba individualmente. Estos se pueden combinar para hacer scripts de prueba más grandes utilizando uno maestro para lograr los escenarios requeridos. Este script maestro se utiliza para invocar los módulos individuales para ejecutar escenarios de prueba de un extremo a otro (E2E).

La razón principal para usar este framework es construir una capa de abstracción para proteger el módulo maestro de cualquier cambio realizado en las pruebas individuales.

|  |  |
| --- | --- |
| Ventajas | Desventajas |
| Mejor escalabilidad y más fácil de mantener debido a la división de la aplicación completa en diferentes módulos. | Toma más tiempo analizar los casos de prueba e identificar flujos reutilizables. |
| Puede escribir scripts de prueba de forma independiente. | Debido a los datos codificados en los scripts de prueba no es posible demandar varios conjuntos de datos. |
| Los cambios en un módulo tienen un impacto nulo o bajo en los otros módulos. | Requiere habilidades de codificación para configurar el framework. |

Tabla 7: Framework de prueba modular

#### Framework de prueba basado en datos

En este framework, mientras que la lógica del caso de prueba reside en los scripts de prueba, los datos de prueba se separan y se mantienen fuera de los scripts de prueba. Los datos de prueba se leen de los archivos externos (archivos de Excel, archivos de texto, archivos CSV, fuentes ODBC, objetos DAO, objetos ADO) y se cargan en las variables dentro del script de prueba. Las variables se utilizan tanto para los valores de entrada como para los valores de verificación. Los scripts de prueba se preparan utilizando Linear Scripting o Test Library Framework.

Ejemplo: UFT, Specflow (C#)

|  |  |
| --- | --- |
| Ventajas | Desventajas |
| Los cambios en los scripts de prueba no afectan los datos de prueba. | Se necesita más tiempo para planificar y preparar tanto los scripts de prueba como los datos de prueba. |
| Los casos de prueba se pueden ejecutar con múltiples conjuntos de datos. |  |
| Se puede ejecutar una variedad de escenarios de prueba simplemente variando los datos de prueba en el archivo de datos externo. |  |

#### Framework de prueba basado en palabras clave

Es conocida como prueba basada en tablas o prueba basada en palabras de acción. El desarrollo de este framework requiere tablas de datos y palabras clave, independientemente de la herramienta de automatización de pruebas utilizada para ejecutarlas. Las pruebas se pueden diseñar con o sin la aplicación. En una prueba basada en palabras clave, la funcionalidad de la aplicación bajo prueba se documenta en una tabla, así como en instrucciones paso a paso para cada prueba.

Aunque trabajar en este framework no requiere muchas habilidades de programación, la configuración inicial (implementarlo) requiere más experiencia.

Ejemplo: Robot Framework (Python).

|  |  |
| --- | --- |
| Ventajas | Desventajas |
| Proporciona una alta reutilización de código | Dado que la inversión inicial es bastante alta, los beneficios de esto solo se pueden obtener si la aplicación es considerablemente grande y los scripts de prueba deben mantenerse durante bastantes años. |
| Herramienta de prueba independiente | Se requiere una gran experiencia en automatización para crear el framework basado en palabras clave. |
| Aunque la aplicación cambie, los scripts de prueba no cambian. |  |
| Las pruebas se pueden diseñar con o sin la aplicación bajo prueba. |  |

Tabla 8: Framework de prueba basado en palabras clave

#### Framework de prueba híbrido

El framework de automatización de pruebas híbridas es la combinación de dos o más frameworks mencionados anteriormente. Intenta aprovechar las fortalezas y los beneficios de otros frameworks para el entorno de prueba particular que administra. La mayoría de los equipos están construyendo este framework impulsado por híbridos en el mercado actual. La industria generalmente utiliza el framework de palabras clave en una combinación con el de descomposición de funciones.

Ejemplo: Karate-DSL, Citrus, etc.

#### Framework de prueba de desarrollo guiado por el comportamiento

El propósito de este framework de desarrollo guiado por el comportamiento es crear una plataforma que permita a todos (como analistas de negocios, desarrolladores, probadores, etc.) participar activamente. Requiere una mayor colaboración entre los equipos de desarrollo y prueba, pero no requiere que los usuarios estén familiarizados con un lenguaje de programación. se utiliza un lenguaje natural no técnico para crear especificaciones de prueba (Gherkin).

Algunas de las herramientas disponibles en el mercado para el desarrollo impulsado por el comportamiento son JBehave (Java) , Cucumber , Specflow (C#), Serenity, etc.

### Beneficios Generales

Estos son los beneficios generales de contar con un framework de prueba:

* Ayuda a reducir el riesgo y los costos.
* Mejora la eficiencia de las pruebas.
* Ayuda a reducir el costo de mantenimiento.
* Permite la reutilización de código
* Permite alcanzar la máxima cobertura de prueba.
* Maximiza la funcionalidad de la aplicación
* Ayuda a reducir la duplicación de casos de prueba
* Ayuda a mejorar la eficiencia y el rendimiento de la prueba con la automatización de la prueba.

## Test Plan

En 1962, la NASA lanzó la Mariner 1 como su primer intento de enviar una nave espacial a Venus. Sin embargo, poco después de su lanzamiento, el cohete se desvió de su curso y se vio obligado a autodestruirse. ¿El costo? $135 millones (en dólares de hoy). ¿La cuestión? Falta de un guion en el código.

Es por eso que antes de que los usuarios tengan acceso a cualquier software o función nueva, debemos ponerlo a prueba a fondo: probar e intentar romperlo. Debemos asegurarnos de que todo lo que hagan los usuarios, responda según lo diseñado. En resumen, necesitamos un plan de prueba o también conocido como test plan.

Un plan de prueba es una de las partes más importantes de cualquier proceso de desarrollo de software. Describe cómo se asegurará de que nuestro producto o función haga lo que se supone que debe hacer y no se rompa cuando los usuarios más lo necesiten.

Pero, ¿qué deberíamos incluir en el plan de prueba? ¿Qué tan profundo debe llegar realmente para asegurarnos de que el producto se mantenga y sus usuarios obtengan lo que esperan?

### Plan de Pruebas

Un plan de prueba es un documento detallado que describe la estrategia de prueba, los objetivos, los recursos necesarios, el cronograma y los criterios de éxito para probar una nueva característica o pieza de software específica.

El objetivo principal, por supuesto, es descubrir defectos, errores y cualquier otra brecha que pueda hacer que el software no actúe como se esperaba o que brinde una mala experiencia a sus usuarios.

#### Objetivos

Un plan de prueba garantiza que el software:

* Cumple con los requisitos que guiaron su diseño y desarrollo, en otras palabras, ¿hace lo que se supone que debe hacer cuando debe hacerlo?
* Responde correctamente a todo tipo de entradas.
* Cumple con los estándares de rendimiento descritas y se puede utilizar según lo previsto.
* Se puede instalar y ejecutar en todos los entornos previstos.
* Logra los resultados que las partes interesadas y yo buscamos.

#### Plantilla del plan de prueba

Cada producto y función tendrá sus propios criterios, estrategias y necesidades de prueba específicos. Además, el objetivo de la prueba cambiará la forma en que lo aborda. Por ejemplo, las pruebas de aceptación del usuario (UAT) son completamente diferentes de las pruebas de estrés y carga, y su plan deberá adaptarse a su objetivo final.

Entonces, ¿qué debería (o podría) incluir en mi plan de prueba? En términos generales, hay algunas áreas principales que querrás incluir y que actuarán como la base de su documento:

1. **Cobertura**: ¿Qué está probando exactamente?
2. **Métodos**: ¿Cómo va a realizar estas pruebas?
3. **Responsabilidades**: ¿Cuáles son los resultados deseados?

##### Cobertura: ¿Qué está pasando exactamente?

Un plan de pruebas debe ser completo, pero no abrumador, es decir, ser específico sobre lo que se incluirá y lo que no.

Después de una breve introducción que destaca los objetivos del plan de prueba, el alcance de alto nivel y el cronograma, deberás definir lo que probará o no probará.

* ¿Qué pruebas va a realizar?
* ¿Por qué ha elegido estos (y no otros)?

##### Métodos: ¿Cómo va a realizar estas pruebas?

A continuación, debe explicar claramente cuál es tu estrategia de prueba. Entre en tantos detalles como sea posible:

* ¿Qué reglas seguirán sus pruebas?
* ¿Qué métricas va a recopilar y a qué nivel?
* ¿Cuántas configuraciones o entornos diferentes va a probar?
* ¿Existen requisitos o procedimientos especiales que deba probar?

También necesitas saber cuándo la prueba fue exitosa. En otras palabras, ¿cuáles son **los criterios de aprobación/reprobación para cada prueba**?

Estos criterios incluyen:

* **Criterio de salida:** ¿cuándo está bien dejar de probar una función y asumir que la función tiene éxito en hacer lo que se propuso hacer?
* **Criterios de suspensión**: ¿cuándo debería pausar una prueba? ¿Existe un umbral de errores en el que debería dejar de realizar pruebas y empezar a buscar soluciones? ¿Cuáles son los pasos para cerrarlo y documentar lo que se ha hecho hasta ahora?
* **Requisitos de reanudación:** ¿cómo se sabe cuándo reanudar una prueba en pausa? ¿Cuáles son los pasos para revisar lo que se ha hecho y retomar?

Por otro lado, es una buena idea en este punto enumerar las suposiciones y riesgos. En otras palabras, ¿qué supone que va a suceder y cuáles son algunos de los riesgos a los que me enfrentaré durante la prueba?

Por último, debes describir las necesidades de recursos y el cronograma del proyecto de prueba. ¿Quién está a cargo de las pruebas y qué recursos necesitan (tanto técnicos como humanos)? ¿Cuándo se realizarán las pruebas y durante cuánto tiempo?

##### Responsabilidades: ¿Cuáles son los resultados deseados?

¿Cuáles son los entregables de prueba requeridos? Esto significa los datos que deseas recopilar, cómo los agruparás en los informes y los problemas y tareas que se devolverán al equipo de desarrollo.

Para asegurarte de que no se pierda nada, cada entrega de prueba debe asignarse a una persona específica de tu equipo en una sección sobre roles y responsabilidades.

### 5 pasos para crear un Test Plan

1. Analizar el producto o la función que estamos probando: Debemos tener un conocimiento profundo del producto o función antes de poder comenzar a crear un plan de prueba para él. Este primer paso es lo que le brinda el contexto para escribir la introducción y los objetivos del plan de prueba y comenzar a planificar los recursos que necesitará para completarlo.
2. Diseñar las estrategias de prueba (y el enfoque) que utilizaremos: Decidir qué probar y documentar en nuestra estrategia de prueba son las partes más críticas del plan de prueba. Lo que se incluye en el alcance de las pruebas dependerá de una serie de factores más allá del producto o la función. Necesitamos profundizar y pensar en:

* Requisito del cliente: ¿qué van a utilizar más nuestros usuarios?
* Presupuesto y cronograma: ¿cuánto tiempo y recursos tenemos para completar las pruebas?
* Especificaciones del producto: ¿cuáles son las partes más importantes de esta función que deben probarse?
* Habilidades del equipo: ¿tenemos la experiencia técnica necesaria para completar cada prueba?

1. Definir los objetivos de la prueba y los criterios de aprobación/reprobación: A medida que definimos cada prueba diferente que se va a ejecutar, necesitamos saber cuándo está terminada. Esto implica definir los criterios de aprobación y reprobación para cada prueba específica, así como algunos ítems mencionados, como los criterios de salida y suspensión.
2. Planificar el entorno de prueba: Los resultados de nuestro plan de prueba dependen tanto de la función que esté probando como del entorno en el que se esté probando. Como parte del alcance, debemos determinar qué combinaciones de hardware, software, sistema operativo y dispositivo deseamos utilizar.
3. Ejecutar el plan de prueba y realizar un seguimiento del progreso: Una vez que nuestro plan de prueba está en su lugar, hay un proceso específico que debemos seguir. Pensemos en esto como el ciclo de vida de las pruebas de software (STLC) que vimos al comienzo de la materia.

### Template: Plan de Testing

#### Introducción

1. **Alcance**: Describir los niveles de testing –por ejemplo, unidad, integración o sistema – y los tipos de testing – como funcional, usabilidad, confiabilidad, desempeño y soportabilidad – que van a ser tratados en este Plan de Test. Es importante indicar cuáles áreas significativas van a ser excluidas del alcance, especialmente aquellas que la audiencia de este documento pueda asumir que son incluidas.

Nota: se debe evitar repetir en esta sección lo que se va a detallar en otras secciones de este documento.

1. **Referencias**: Esta sección provee un completo listado de todos los documentos utilizados para la confección de este documento. Especificar la fuente de donde estos documentos son obtenidos.
2. **Definiciones y abreviaturas**: Esta sección proveerá las definiciones de todos los términos, siglas y abreviaciones requeridas para interpretar adecuadamente este documento.

#### Objetivos del test

Proveer una visión general del objetivo y motivación de las pruebas que serán dirigidas en esta iteración

1. **Antecedentes**: Incluir una breve descripción del entorno dentro del cual se emprenderá el esfuerzo de testing definido por este Plan de Test. Proveer información acerca del problema a ser resuelto, los mayores beneficios de la solución, la arquitectura planeada y una breve historia del proyecto. Cuando esta información esté definida en otros documentos, puede incluir referencias a estos. Esta sección debería incluir entre tres y cinco párrafos.
2. **Objetivo de la Evaluación:** Definir brevemente el objetivo del esfuerzo de evaluación en la iteración actual.

* Esta definición puede incluir uno o más intereses como:
* Encontrar tantos errores como sea posible.
* Encontrar problemas importantes, evaluar riesgos de calidad percibidos.
* Obtener información sobre riesgos del proyecto percibidos.
* Certificar un estándar.
* Verificar una especificación (requerimientos, diseño o afirmaciones).
* Obtener información acerca de la calidad del producto y la satisfacción de los interesados.
* Obtener información sobre el testing.
* Cumplir con las obligaciones del proceso.

Cada objetivo provee un contexto diferente al esfuerzo de testing y altera la manera en que el testing será abordado.

1. **Motivaciones del Test: Incluir** una descripción de los elementos principales que motivaron el esfuerzo de testing de esta iteración. El testing será motivado por varios elementos como riesgos de calidad, riesgos técnicos, riesgos del proyecto, casos de uso, requerimientos funcionales y no funcionales, elementos de diseño, defectos sospechados o fallas, pedidos de cambio, entre otros.

#### Items a testear

Especifica el nivel de testing al que se hace referencia en el documento (sistema, integración, unidad). Lista de las propiedades funcionales y técnicas a testear, con referencia a los documentos de donde se obtiene la información. Se identifican los ítems que serán excluidos de las tareas de testing.

Proveer una lista de alto nivel de los ítems principales que serán objeto de test. Esta lista debería incluir tanto los ítems producidos por el equipo de desarrollo del proyecto como los ítems de los que dependen; por ejemplo, el procesador, los equipos periféricos, los sistemas operativos, los productos de terceras partes y componentes. Considere agrupar la lista por categorías y asignar una importancia relativa a cada motivación del test.

#### Métodos

Describe cómo se llevarán a cabo los tests. Especifica las actividades principales, técnicas y herramientas que deben ser usadas durante el testing. Especifica los criterios de selección de casos con los cuales se eligieron los casos presentados en los casos de pruebas. Especifica los criterios de terminación de la actividad.

#### Casos de Pruebas

**Estrategia**: criterio utilizado para la selección de casos (por ejemplo, análisis de casos de borde, partición en clases, casos de uso, aspectos de la aplicación, etc.); factores a tener en cuenta.

**Casos**: presenta la lista de casos para cada ítem. Presenta, además, la progresión de los casos, es decir, el orden en el cual deben testearse los casos identificados.

Cada caso debe ser descrito en términos de los parámetros de entrada (condiciones sobre los parámetros que van a ser testeadas), considerando situaciones válidas e inválidas; para cada caso de test definido se debe precisar el resultado esperado (funcionamiento correcto del sistema) y qué requerimiento (funcional o no funcional) se está ejercitando (de los definidos en los ítems a testear).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Número | Descripción | Tipo | Requerimiento | Resultado Esperado |
| Numeración secuencial de los casos. | Descripción de las condiciones de entrada que determinan el caso. | Si corresponde a un caso válido o inválido. | Requerimiento (funcional o no funcional) ejercitado por el caso (de los descritos anteriormente). | Qué debería devolver el sistema (mensajes, transformaciones a la base de datos, listados, cálculos, errores, etc.). |

Tabla 9: Tabla modelo para la documentación de casos

#### Criterios de aceptación

###### Plan de Test

1. **Criterio de Entrada del Plan de Test**: Especificar el criterio que será utilizado para determinar cuando la ejecución del Plan de Test puede comenzar.
2. **Criterio de Salida del Plan de Test**: Especificar el criterio que será utilizado para determinar cuando la ejecución del Plan de Test está completa o cuando la ejecución ya no provee ningún beneficio.
3. **Criterio de Suspensión y Reanudación:** Especificar el criterio que será utilizado para determinar si el testing debe ser suspendido o terminado prematuramente antes de que el plan se haya ejecutado totalmente, y bajo qué criterio el testing puede ser reanudado.

###### Ciclo de Test

1. **Criterio de Entrada del Ciclo de Test**: Especificar el criterio que será utilizado para determinar cuando el esfuerzo de test para el siguiente Ciclo de Test de este Plan de Test puede comenzar.
2. **Criterio de Salida del Ciclo de Test:** Especificar el criterio que será utilizado para determinar cuando el esfuerzo de test para el Ciclo de Test actual de este Plan de Test es considerado suficiente.
3. **Terminación Anormal del Ciclo de Test:** Especificar el criterio que será utilizado para determinar cuando el testing del Ciclo de Test actual debe ser prematuramente suspendido o terminado, o cuando la versión candidata debe ser alterada.

#### Entregables

En esta sección se enumeran los artefactos que serán creados por el esfuerzo de testing que son útiles a los interesados. No listar todos los productos, solo aquellos que proporcionan un beneficio directo y tangible a los interesados y aquellos que se deseen utilizar para medir el éxito del esfuerzo.

1. **Reporte de Cubrimiento del Testing:** Incluir una breve reseña de la forma y del contenido de los informes usados para medir el avance del testing, e indicar con qué frecuencia serán producidos. Informar el método y las herramientas usadas para registrar, para medir, y para divulgar el avance.
2. **Log de Incidentes y Pedidos de Cambio:** Incluir una breve reseña del método y herramientas utilizadas para registrar, seguir y administrar los incidentes de test, los pedidos de cambio asociados y su estado.
3. **Scripts de Test:** Incluir una breve reseña de los elementos que serán entregados para permitir la ejecución de Testing de Regresión sobre versiones posteriores de la aplicación para ayudar a detectar regresiones en la calidad del producto.
4. **Entregables Adicionales:** En esta sección se deben identificar aquellos productos que son entregables opcionales o aquellos que no deben ser utilizados para medir o evaluar el éxito de la ejecución del Plan de Test.
5. **Scripts Automatizados Adicionales:** Estos pueden ser una colección de código fuentes de los scripts automatizados de test, o un repositorio de los códigos fuentes y ejecutables mantenidos en un software de automatización de test.

#### Recursos Requeridos

Recursos requeridos para llevar a cabo el testing correspondiente a este plan, en cuanto a hardware, software (incluyendo seteos de datos, configuración de la base, etc.) y personal involucrado (es aquí donde se asigna personal de acuerdo a los roles establecidos en la Estrategia de Testing).

#### Responsabilidades

Describe quién:

* Autoriza que el testing pueda comenzar.
* Lleva a cabo los tests.
* Autoriza la terminación de las actividades de testing.

Si bien estas actividades serán documentadas en los entregables correspondientes a la ejecución del testing, es necesario definir responsabilidades durante el plan para poder realizar una planificación real de las tareas.

#### Planificación temporal de áreas

Estimación de los tiempos requeridos para las distintas tareas planificadas, incluyendo el armado del ambiente, selección de datos de test, y otras tareas relacionadas. Puede incorporar un Gantt detallando la duración de las tareas, dependencias y responsables.

Clase 26: DevOps Tester

## ¿Qué es DevOps?

Es un conjunto de prácticas que combina el desarrollo de software (Dev) y las operaciones de IT (Ops), lo que ayuda a acortar el ciclo de vida del desarrollo de sistemas y proporciona integración y entrega continuas con software de alta calidad.

DevOps detalla el proceso de desarrollo y entrega de un software, incluyendo su producción completa de extremo a extremo, desde el concepto del producto hasta la satisfacción del cliente, por lo que se debe entender que el software no está completo hasta que el usuario lo acepte.

En un entorno de DevOps, el software se crea, se prueba, se lanza y se implementa con mayor rapidez, frecuencia y confiabilidad que con otros enfoques.

Para implementar este concepto utilizamos un pipeline. Este consiste en un flujo comprendido en varias fases que van en forma secuencial, siendo la entrada de cada una la salida de la anterior.

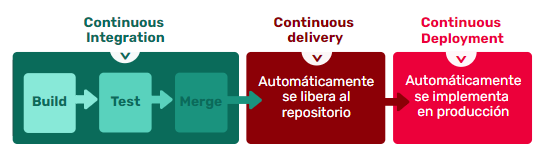
El pipeline estaría formado por un conjunto de procesos o herramientas automatizadas que permite que los desarrolladores, como otros roles, trabajen de forma coherente para crear e implementar código en un entorno de producción.

Este set de prácticas de DevOps está integrado por las siguientes:

* Continuous Integration (CI) - Integración Continua
* Continuous Delivery (CD) - Entrega Continua
* Continuous Deployment (CD) - Implementación Continua

La CI/CD (nominación que incluye las prácticas nombradas) incorpora la automatización continua y el control permanente en todo el ciclo de vida de las aplicaciones, desde las etapas de integración y prueba hasta las de distribución e implementación.

A continuación, se detalla de qué trata cada una:



### Continuous Integration

Los desarrolladores que practican la integración continua mergean sus cambios con la rama principal con la mayor frecuencia posible. Los cambios del desarrollador se validan creando una compilación y ejecutando pruebas automatizadas contra la compilación.

Este enfoque asegura la detección temprana de errores y minimiza los problemas de integración de código.

La integración continua pone un gran énfasis en la automatización de pruebas para verificar que la aplicación no se interrumpa cada vez que se integran nuevos commits en la rama principal.

### Continuous delivery

La entrega continua consiste en automatizar el proceso de lanzamiento para que la aplicación se pueda implementar automáticamente en cualquier momento.

Se refiere a los cambios que implementa un desarrollador en una aplicación, a los que se les realizan pruebas de errores automáticas y que se cargan en un repositorio, para que luego el equipo de operaciones pueda implementarlos en un entorno de producción. Es una solución al problema de la poca visibilidad y comunicación entre los equipos comerciales y de desarrollo. Con la entrega continua puede decidir publicarlo diariamente, semanalmente, quincenalmente o lo que se adapte a los requisitos de su negocio

Con ese fin, el propósito de la entrega continua es garantizar que la implementación del código nuevo se lleve a cabo con el mínimo esfuerzo.

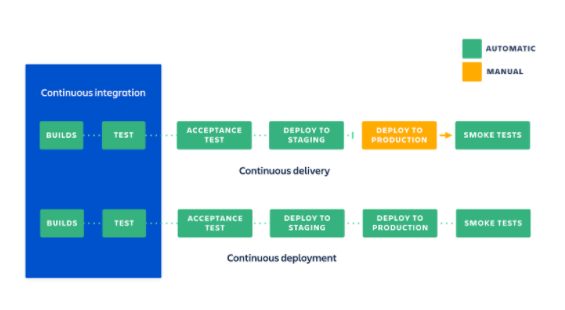
### Continuous Deployment

La implementación continua va un paso más allá que la entrega continua.

Es un proceso de lanzamiento de software que utiliza pruebas automatizadas para validar si los cambios en una base de código son correctos y estables para la implementación autónoma inmediata en un entorno de producción.

Hace referencia a la liberación automática de los cambios que implementa el desarrollador desde el repositorio hasta la producción, para que los clientes puedan usarlos. Así se resuelve el problema de sobrecargar a los equipos de operaciones con procesos manuales que retrasan la distribución de las aplicaciones.

### Relación entre las prácticas CI/CD



Desde el lado de testing se deben crear y seleccionar paquetes rápidos y confiables de pruebas automatizadas que se ejecuten como parte del pipeline.

Cada cambio genera una compilación que crea paquetes de software; se ejecutan pruebas de unidades y, quizá, se realizan otras verificaciones, como el análisis estático. Una vez que estos paquetes pasan la primera etapa, se ejecutan pruebas de aceptación automatizadas más completas. También se suelen ejecutar algunas pruebas no funcionales, como pruebas de rendimiento y análisis de vulnerabilidades, en el software en ejecución implementado de forma automática. Por lo general, las compilaciones que pasan la etapa de aceptación luego están disponibles para la exploración manual y las pruebas de usabilidad. Por último, si no se encuentran errores en estos pasos manuales, se considera que la app se puede lanzar.

La implementación de estas prácticas permite a los equipos centrarse en la creación de código y elimina la carga y la posibilidad de errores humanos en los pasos cotidianos que se realizan manualmente.

### Herramientas

Existen diversas herramientas que permiten la implementación de estas prácticas. A continuación, se nombran las más conocidas:

* **Jenkins**: es un servidor de automatización de código abierto y gratuito que ayuda a automatizar los procesos de desarrollo de software, como la construcción, facilitando el CI/CD, el lanzamiento y los testeos.
* **Docker**: ayuda a los desarrolladores a construir, empaquetar y luego desplegar los códigos con facilidad y rapidez a través de contenedores con las dependencias necesarias en lugar de máquinas virtuales.
* **Puppet**: esta herramienta de automatización de DevOps ayuda a gestionar las diferentes etapas del ciclo de vida del software, como el aprovisionamiento de la infraestructura de IT, la aplicación de parches y la configuración y gestión de los componentes de software y el sistema operativo en las infraestructuras de la nube y los centros de datos.
* **CircleCI**: ejecuta su canalización CI/CD automáticamente en una máquina virtual o un contenedor limpio para facilitar las pruebas.
* **Bamboo de Atlassian**: permite la integración continua, el despliegue continuo y la entrega de aplicaciones de software. Se pueden unir las compilaciones, lanzamientos y pruebas automatizadas como un solo flujo de trabajo.
* **TeamCity**: es una herramienta de CI/CD DevOps de uso general que ofrece mayor flexibilidad para diferentes tipos de prácticas de desarrollo y flujo de trabajo. Se pueden crear, desplegar y probar aplicaciones, contenedores y paquetes de diferentes tipos, ya sea en varias nubes, en varios idiomas o en varias plataformas.
* **Herramientas de DevOps en la nube**: Kamaterra, OpenStack, AWS, Google Cloud Platform, Azure, IBM Cloud, etc.

## ¿Quiénes participan en las pruebas de DevOps?

Los testers o probadores del equipo deben aprovechar las mejores prácticas en pruebas ágiles, integración continua y desarrollo guiado por pruebas para acelerar sus procesos de control de calidad de pruebas y reducir el tiempo del ciclo. Un aspecto fundamental de las DevOps pipelines es la noción de responsabilidad compartida para lograr la máxima calidad. En equipos ágiles, todos son igualmente responsables de la calidad del producto o del éxito del proyecto. Esto significa que las pruebas las realiza todo el equipo, no solo los probadores designados o los profesionales de control de calidad, incluidos los miembros del equipo cuya experiencia principal puede ser en programación, análisis comercial, base de datos o administración de sistemas. Los miembros del equipo que tienen mayor experiencia en las pruebas de DevOps o en el uso de una herramienta de prueba en particular no se limitan a realizar solo esa actividad; también pueden colaborar con los clientes o stakeholders en los requisitos del producto y trabajar con otros miembros del equipo para desarrollar código de alta calidad que cumpla con esos requisitos.

Los probadores involucrados con actividades de DevOps deben tener habilidades de trabajo flexibles que les permitan moverse cómodamente de un área de DevOps a otra, ya sea en desarrollo, integración, prueba, lanzamiento o implementación. Comenzar con una comprensión de las personas, la cultura y cómo funciona la organización también ayuda si tienen fuertes habilidades de comunicación. La estrategia de prueba de DevOps que se construye se centra en simplificar el desarrollo general, las pruebas y el entorno operativo para lograr el objetivo de la entrega continua.

Entre las habilidades técnicas que necesitan los testers/probadores que están relacionadas con DevOps se encuentra la comprensión de algunas de las siguientes herramientas y tecnologías:

* Control de código fuente (usando Git, Bitbucket, Svn, VSTS, etc.).
* Integración continua (usando Jenkins, Bamboo, TeamCity, VSTS).
* Automatización de infraestructura (usando Puppet, Chef, Ansible).
* Automatización y organización de la implementación (con Jenkins, VSTS, Octopus Deploy).
* Conceptos de contenedor (LXD, Docker).
* Orquestación (Kubernetes, Mesos, Swarm).
* Nube (con AWS, Azure, Google Cloud, Openstack).

Otra habilidad útil es la capacidad de escribir código de secuencias de comandos en lenguajes como JavaScript, Python, Perl o Ruby para implementar tecnologías y herramientas de automatización en cualquier nivel, desde los requisitos hasta el desarrollo, las pruebas y las operaciones.