Zoé Agustina Tira

[Nombre de la empresa]  [Dirección de la compañía]

testing

[Módulo 1 - Fundamentos de testing y Gestión de Defectos 2](#_Toc86934109)

[C1A – PRIMEROS PASOS 2](#_Toc86934110)

[C2A – GESTIÓN DE DEFECTOS 4](#_Toc86934111)

[Módulo 2 - Diseño e implementación de la prueba 7](#_Toc86934112)

[C4A – DISEÑO DE LA PRUEBA 7](#_Toc86934113)

[C5A – NIVELES Y TIPOS DE PRUEBA 10](#_Toc86934114)

[C7A – TÉCNICAS DE PRUEBA 22](#_Toc86934115)

[C8A – IMPLEMENTACIÓN Y EJECUCIÓN DE LA PRUEBA 25](#_Toc86934116)

[Módulo 3 - Análisis y Planificación de la prueba 26](#_Toc86934117)

[C10A – PRUEBAS ESTÁTICAS Y DINÁMICAS 26](#_Toc86934118)

[Módulo 4 - Análisis y Planificación de la prueba 30](#_Toc86934119)

[C11A – ORGANIZACIÓN DE LA PRUEBA 30](#_Toc86934120)

[Módulo 4 – Unit Testing 34](#_Toc86934121)

[C13A – INTRODUCCIÓN AL DEBUGGING 34](#_Toc86934122)

[C14A – INTRODUCCIÓN A LA PRUEBA DE COMPONENTE 38](#_Toc86934123)

[C16A – Prueba de componente 41](#_Toc86934124)

[C17A – Test unitario 45](#_Toc86934125)

[Módulo 6 – Be testing 52](#_Toc86934126)

[C19A – Introducción a api testing 52](#_Toc86934127)

[C20A– API Testing 53](#_Toc86934128)

[Módulo 7 - Introducción a Automation 65](#_Toc86934129)

[C22A - Fundamentos de automatización de la prueba 65](#_Toc86934130)

[C23A – Automatización de la prueba 71](#_Toc86934131)

# Módulo 1 - Fundamentos de testing y Gestión de Defectos

## C1A – PRIMEROS PASOS

Introducción al testing

Un tester parte siempre de la suposición que un programa contiene errores. Por lo que deberá probarlo para encontrar la mayor cantidad y avisar para que lo soluciones. Brindan calidad a ese desarrollador. Testing = Calidad.

*Calidad = Satisfacción del cliente. Testing = Confianza.*

Realizar pruebas frecuentes es vital en el proceso de desarrollo.

Existen proyectos en la fase final, que tienen que volver a corregirse por factores humanos, frecuentemente debido a:

* Requerimientos poco claros.
* Modificaciones a último momento.
* Errores de diseño y testers no calificados.

El testing sirve para:

* Encontrar el mayor número de defectos y remediarlos.
* Asegurar el buen funcionamiento del producto.
* Lograr un mayor grado de calidad.

7 principios de testing:

*1) 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.

*2) 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.

*3) 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.

*4) 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. Tratar de no tener que probar todas las posibilidades, agrupamos comportamientos. Agrupar errores arrastrados.

*5) 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.

*6) 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.

*7) 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.

Rol del tester

Testing: Proceso de ejecución de un programa con la intención de encontrar errores.

Aspecto psicológico: Los humanos tienden a ser orientados a objetivos y establecimientos de metas. En base a lo que queremos demostrar es el camino que seguiremos y puede traer subconscientemente menos probabilidad de demostrar lo contrario a nuestro pensamiento.

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.

Puede ser diferente en desarrollo ágil, pueden los propietarios del producto realizar prueba de aceptación para validar las historias de usuario al final de cada iteración, etc.

Beneficios:

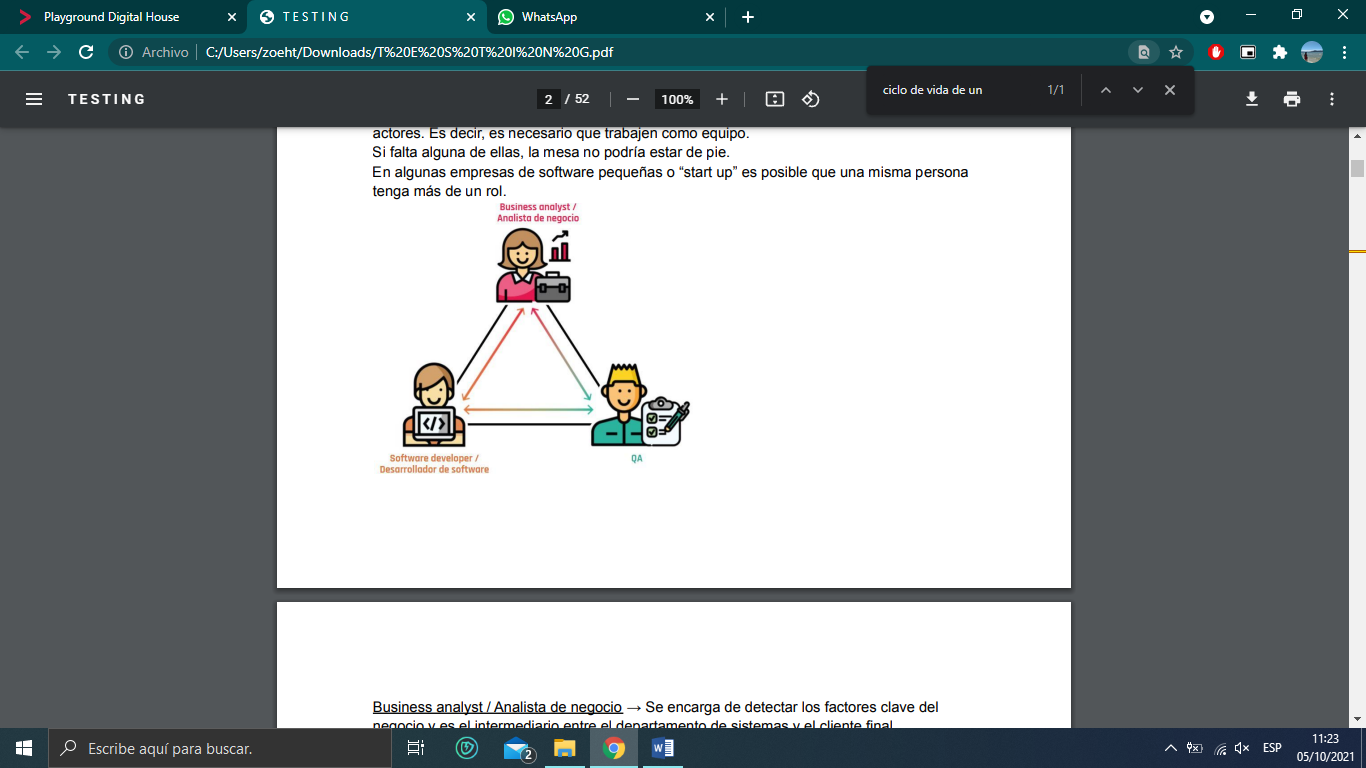
* Reconocimiento de fallos debido a diferente contexto, perspectivas técnicas y sesgos.
* Verificar, cuestionar o refutar las suposiciones hechas por implicados durante la especificación e implementación del sistema.

Desventajas:

* Pueden perder el sentido de la responsabilidad respecto a la calidad.
* Desarrolladores independientes pueden ser vistos como un cuello de botella o ser culpados por retrasos o liberación.
* Probadores independientes pueden carecer de información importante, por ejemplo el objeto de prueba.

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. 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.



## C2A – GESTIÓN DE DEFECTOS

Gestión de defectos

Ciclo de vida de un defecto: El proceso que gestiona un defecto desde su descubrimiento hasta su solución se denomina ciclo de vida de un defecto.

En cada estado solo existe un responsable del defecto, excepto en estados terminales -cerrado, duplicado-, debido a que no se van a realizar más acciones.

|  |  |  |
| --- | --- | --- |
| **Cerrado**  El defecto fue corregido y se encuentra disponible para el usuario final. | 1. **Nuevo/Inicial**   Se recopila la información y se registra el defecto. | **Asignado**  De ser un defecto validado y debe solucionarse asigna al equipo de desarrollo, sino se puede rechazar o diferir.  (en casos como duplicado, diferido, devuelto o rechazado) |
| **Verificado**  Se obtiene el resultado esperado en la prueba de confirmación. |  | **El proceso**  Se analiza y trabaja en la solución. |
| **En verificación**  El probador ejecuta una prueba de confirmación.  Puede reabrirlo, indicando que el defecto no se ha solucionado. | **En espera de verificación**  Espera que sea asignado a un probador. El desarrollador está a la expectativa del resultado de la verificación. | **Corregido**  Se realizan los cambios en el código para solucionar el defecto. |

Error, defecto y falla: Un error es cometido por un programador, esto genera un defecto en el software, lo que culmina en un fallo del sistema.



Proceso de gestión de defectos:

1. Detectar
2. Registrar (varía según el contexto del componente/sistema/nivel de prueba/modelo de desarrollo elegido.
3. Investigación y seguimiento
4. Clasificación/Resolución

*Objetivos:*

1. Brindar Información sobre eventos adversos, identificar efectos específicos, aislar problema con una prueba de producción mínima y corregir defectos potenciales.
2. Proporcionar a jefes de prueba un medio para hacer seguimiento de calidad del producto y del impacto en la prueba.
3. Dar ideas para la mejora de procesos de desarrollo y prueba.

Escribir un buen informe:

1. Si el defecto se reporta eficientemente, aumenta probabilidades de solucionarlo rápida y efectivamente.
2. Los bugs deben tener identificadores únicos (ID)
3. Las fallas deben ser reproducibles para reportarla, de no serlo, no es un defecto. Se puede hacer una nota personal para posterior evaluación.
4. Especificidad, no debe haber suposiciones o ideas sobre lo que ocurre u otra información irrelevante para reproducir el defecto.
5. Reportar cada paso para reproducirlo.

*Problemas más comunes al escribir informes:*

1. Redactar defecto de manera excesivamente coloquial y ambigua.
2. Dar solo una captura del defecto sin indicar contexto
3. No incluir en descripción del defecto cual era el resultado esperado.
4. No determinar un patrón con el cual el defecto ocurre. -es importante para ser directos en cuál es el problema.-
5. No leer el defecto reportado siguiendo los pasos nosotros mismo para ver que la descripción es clara.
6. No incluir información que dada las características del defecto, la misma es de relevancia.

Cuando se detecta un defecto —como parte de las pruebas estáticas—, o se observa un fallo —como parte de las pruebas dinámicas—, la persona implicada debería recopilar los datos e incluirlos en el informe de defectos. Esta información debería ser suficiente para tres fines:

* Gestión del informe durante el ciclo de vida de los defectos.
* Evaluación del estado del proyecto, especialmente en términos de calidad del producto y progreso de las pruebas.
* Evaluación de la capacidad del proceso.

Los datos necesarios para la gestión de los informes de defectos y el estado del proyecto pueden variar en función de cuándo se detecta el defecto en el ciclo de vida, siendo la información requerida menor en etapas anteriores —por ejemplo, revisiones de requisitos y pruebas unitarias—. No obstante, la información básica recopilada debería ser coherente durante todo el ciclo de vida e, idealmente, en todos los proyectos.

Partes de un informe de defectos:

*Id* – Identificador, código único e irrepetible.

*Titulo* – Corto y específico, que se entienda lo que queremos reportar. Cuando el desarrollador o el equipo vean el titulo deben poder interpretar rápidamente qué es, dónde esta y cuán importante es el defecto.

*Descripción* – Describir un poco más el error.

*Fecha del informe del defecto* – Cuándo se detectó el defecto.

*Autor* – Quién detecto el defecto.

*Identificación del elemento de prueba* – Nombre de la aplicación o componente que estamos probando.

*Versión* – Numero que indica en que versión esta la aplicación.

*Entono* – Entorno en que probamos (desarrollo, QA, producción)

*Pasos a reproducir* – Pasos a seguir para llegar al defecto.

*Resultado esperado* – Lo que se espera que suceda, muchas veces según los requerimientos.

*Resultado obtenido o actual* – Lo que realmente sucedió, puede o no coincidir con lo esperado, si no coincide hemos detectado un error o bug.

*Severidad* – Cuan grave s el defecto que encontramos: bloqueado, crítico, alto, medio, bajo o trivial.

*Prioridad* – Qué tan rápido debe solucionarse el defecto: alta, media, baja.

*Estado del defecto* – Nuevo, diferido, duplicado, rechazado, asignado, ne proceso, corregido, en espera de verificación, en verificación, verificado, reabierto y cerrado.

*Referencias* – Link al caso de prueba con el cual encontramos el error.

*Imagen* – Opcional, captura de pantalla del error.

# Módulo 2 - Diseño e implementación de la prueba

## C4A – DISEÑO DE LA PRUEBA

Caso de prueba

Conjunto de acciones que se ejecutan para verificar una característica o funcionalidad particular de una aplicación de software. Documento escrito que proporciona información escrita sobre qué y cómo probar. Es por ello que es de vital importancia en el mundo de la calidad.

Características de un buen caso de pruebas.

1. 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.
2. Asegurar la mayor cobertura posible – Escribir casos de prueba para todos los requisitos especificados.
3. Autonomía – El caso de prueba debe generar los mismos resultados siempre, sin importar quien lo pruebe.
4. Evitar la repetición de caos de prueba – Si se necesita un caso de prueba para ejecutar otro, indicar el caso de prueba por su ID.
5. 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 ejecución sea más rápida.
6. El titulo debe ser fuerte – Solo leyendo el título, cualquier probador debería comprender el objetivo del caso de prueba.
7. 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?

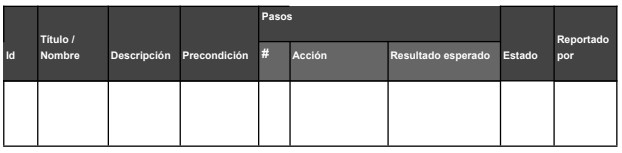
*Identificador* – Numérico o alfanumérico, por lo gral se generan automáticamente.

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

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

*Precondición* – Cosas que deben cumplirse antes de ejecutar el caso de prueba.

*Pasos* – Acciones que deben realizarse para obtener los resultados.

*Resultados esperados* – Lo que le indica al probador cual debería ser la experiencia luego de ejecutar los pasos y determinar si el test fallo o no. 

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 a 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 cómo 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 a ahorrar esfuerzos posteriores.

Limitaciones:

* 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.

Casos de uso - Caso de prueba

Antes de realizar el diseño de los casos de prueba, lo que se debe llevar a cabo es el análisis de los documentos que van a ser la base para la generación de esos casos de prueba. Estos documentos van a asegurar los requisitos del cliente.

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.

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.

## C5A – NIVELES Y TIPOS DE PRUEBA

Ciclo de vida de las pruebas de software (STLC)

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

Proceso de prueba en contexto:

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

Plazos.

Complejidad.

Tareas principales – aunque no están siempre 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
8. Diseño.

Planificación:

* Definen objetivos y enfoque de la prueba dentro de restricciones impuestas por contexto.
* 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 limite
* Generar el plan de pruebas.

*Documentos de salida:*

* Plan de prueba – general y/o por nivel de prueba-.

Seguimiento y control: Reunir información y proporcionar retroalimentación y visibilidad sobre las actividades de la 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.

*Subactividades:*

* Comprobar resultados y 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: Determina “que probar”.

*Subactividades:*

* Analizar 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 riesgo, 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.
* Capturar la trazabilidad entre la base de prueba y las condiciones de prueba.

*Documento de salida:*

* Contrato de prueba que contiene las condiciones de la prueba.

Diseño: Se determina “como probar”

*Subactividades:*

* 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 entre la 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.

*Subactividades:*

* 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, 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.

*Subactividades:*

* Registrar los identificadores y las versiones de los elementos u objetos de prueba.
* Ejecutar y registrar el resultado de las 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 prueba y los resultados de la prueba.

*Documento de salida:*

* Reporte de defectos.
* Informe de ejecución de prueba.

Conclusión:

*Subactividades:*

* 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 la prueba para su posterior reutilización.
* Traspaso de los productos de prueba a otros equipos que podrían beneficiarse con su uso
* Analizar las elecciones 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.

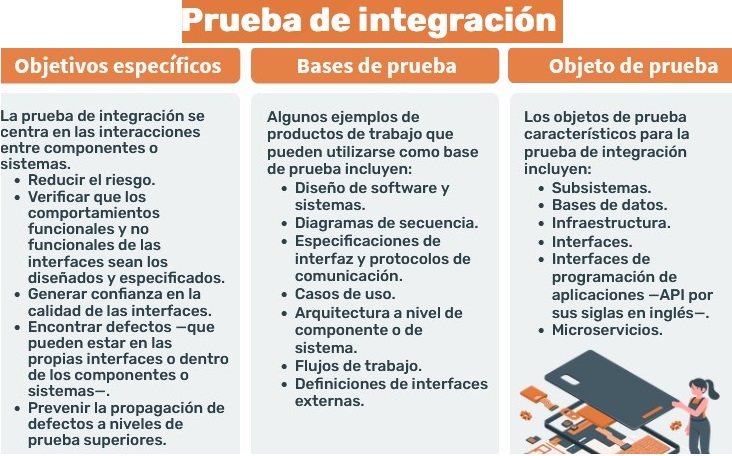
Niveles de prueba:

Prueba unitaria o de componente:



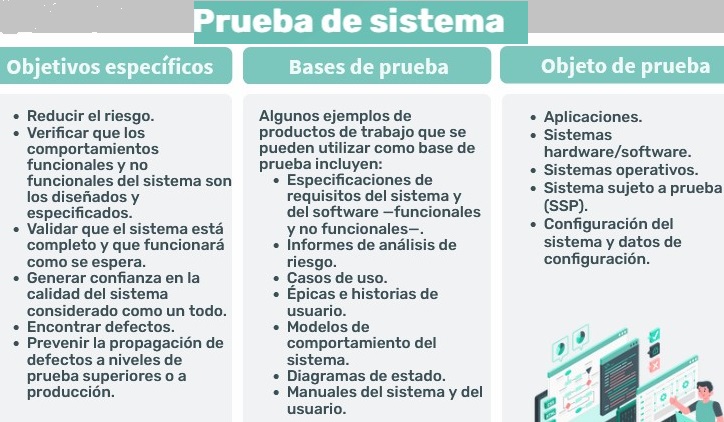


Prueba de integración:





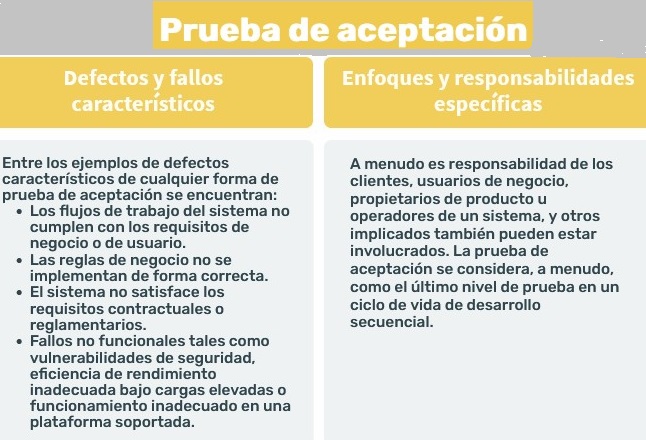
Prueba de sistema:





Prueba de aceptación:





|  |  |
| --- | --- |
| **Desde el análisis a la implementación** | |
| 1. Etapas de procesos de larga duración y secuenciales. | 1. Iteraciones cortas y definidas como sprints. |
| 1. 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. | 1. Pruebas dinámicas y continuas durante la iteración. Los requerimientos son analizados como presentaciones (futuras) y se crean escenarios de prueba usando un enfoque de desarrollo guiado por comportamiento (BDD). |
| 1. Se realiza una prueba de ambigüedad sobre los requerimientos. | 1. Se dividen las prestaciones en historias de usuarios (user stories) y se modelan escenarios de prueba para esas historias. |
| 1. Se dividen los requerimientos en historias pequeñas. | 1. Se escriben los unit test y luego se codifica la solución usando un enfoque de desarrollo guiado por pruebas (TDD). |
| 1. Se desarrolla el código en base a la documentación de diseño. Luego de tener el código se desarrollan los unit test. | 1. Se enfoca en la ligera documentación priorizando lo que entregue valor al cliente. |
| 1. Existe mayor documentación de cada proceso. | 1. Se ejecutan pruebas continuamente y desde etapas tempranas. Toma relevancia la aplicación de integración continua (CI) y distribución o desarrollo continuo (CD). |
| 1. Las pruebas se ejecutan luego de tener el código generado de todo un requerimiento o todo el sistema. | 1. Adquieren gran protagonismo las pruebas automatizadas. |
| 1. las pruebas son generalmente manuales. | 1. Además, la integración continua permite tener una retroalimentación activa. |

Tipos de pruebas

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, basadas en objetivos de pruebas específicas.

* [Prueba funcional.](#PruebaFuncional)
* [Prueba no funcional.](#PruebaNoFuncional)
* [Pruebas estructurales.](#PruebasEstructurales)
* [Prueba asociada al cambio.](#PruebaAsociadaAlCambio)

*Dichos objetivos pueden incluir:*

1. Evaluar las características de calidad funcional tales como la completitud, corrección y pertinencia.
2. Evaluar características no funcionales de calidad, tales como la fiabilidad, eficiencia de desempeño, seguridad, confiabilidad y usabilidad.
3. Evaluar si la estructura o arquitectura del componente o sistema es correcta, completa y según lo especificado.
4. Evaluar los efectos de los cambios, tales como confirmar que los defectos han sido corregidos (prueba de confirmación) y buscar cambios no deseados en el comportamiento que resulten de los cambios en el software o en el entorno (prueba de regresión)

Prueba funcional:

|  |  |
| --- | --- |
| **Definición** | La prueba funcional de un sistema incluye pruebas que evalúan las funciones que el sistema debe realizar. Las funciones describen qué hace el sistema. |
| **Implementación** | La prueba funcional observa el comportamiento del software. |
| **Niveles de Prueba** | Se pueden realizar pruebas funcionales en todos los niveles de prueba. |
| **Alcance** | Los requisitos funcionales pueden estar detallados en los siguientes documentos: especificaciones de requisitos del negocio, épicas, historias de usuarios, casos de uso y/o especificaciones funcionales. |
| **Cobertura** | La cobertura funcional es la medida en que algún tipo de elemento funcional ha sido practicado por pruebas, y se expresa como un porcentaje del tipo o tipos de elementos cubiertos. |

Prueba no funcional:

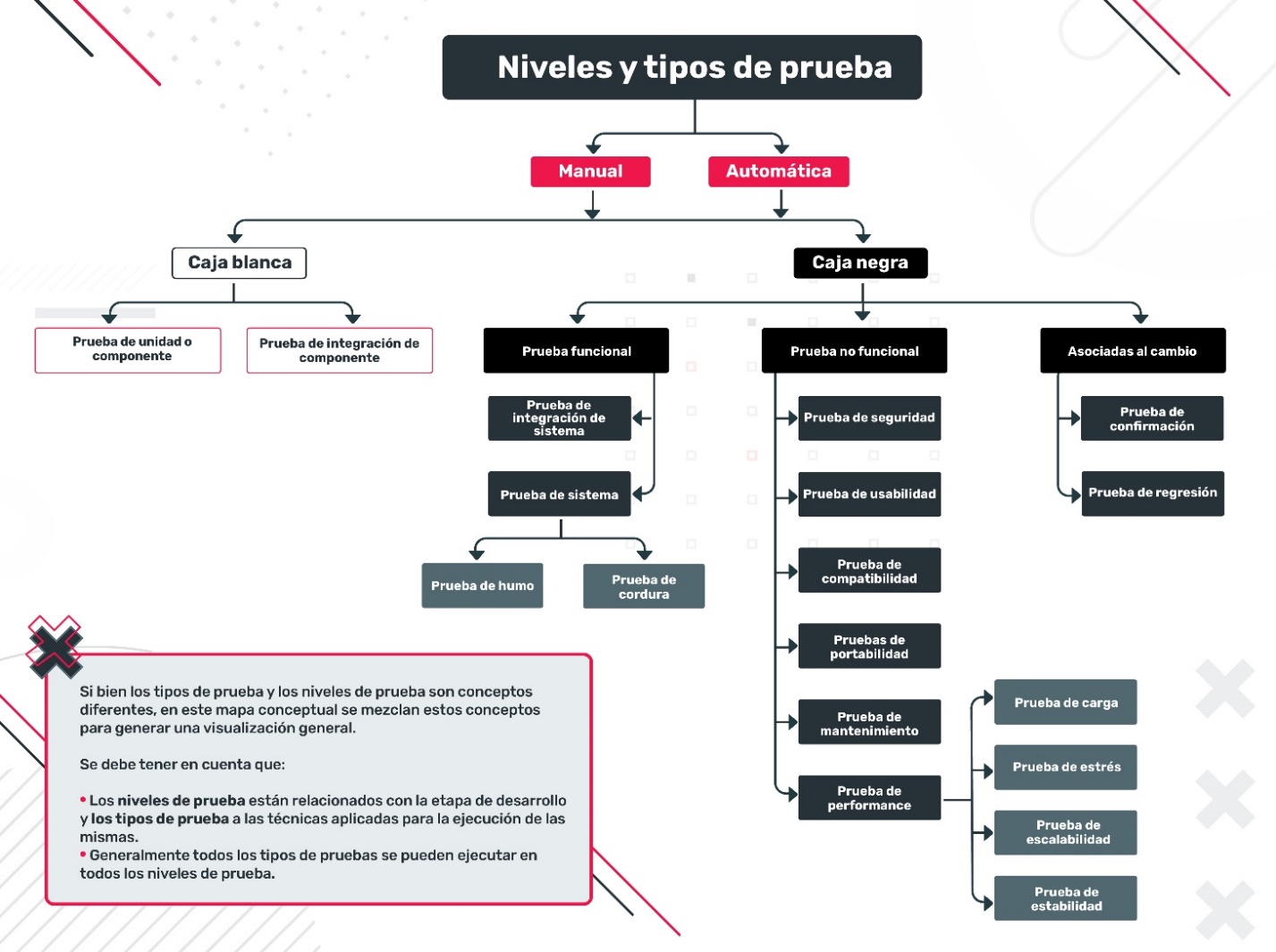
|  |  |
| --- | --- |
| **Definición** | La prueba no funcional prueba “**cómo de bien**” se comporta el sistema. |
| **Implementación** | El diseño y ejecución de la prueba no funcional puede implicar competencias y conocimientos especiales, como el conocimiento de las debilidades inherentes a un diseño o tecnología -por ejemplo: vulnerabilidades de seguridad asociadas con determinados lenguajes de programación-. |
| **Niveles de Prueba** | Se pueden realizar pruebas no funcionales en todos los niveles de prueba |
| **Alcance** | La prueba no funcional del sistema evalúa características como la usabilidad, la eficiencia del desempeño o la seguridad. |
| **Cobertura** | La cobertura no funcional es la medida en que algún tipo de elemento no funcional ha sido practicado por pruebas,  y se expresa como un porcentaje del tipo o tipos de elementos cubiertos. |

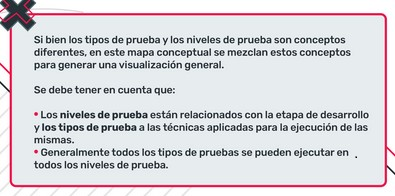
Pruebas estructurales:

|  |  |
| --- | --- |
| **Definición** | Estas pruebas están basadas en la estructura interna del sistema o en su  implementación. La estructura interna puede incluir código, arquitectura, flujos de trabajo y/o flujos de datos  dentro del sistema |
| **Implementación** | El diseño y la ejecución de este tipo de pruebas pueden implicar competencias o conocimientos  especiales, como la forma en que se construye el código, cómo se almacenan los datos, y cómo utilizar las herramientas de cobertura e interpretar correctamente sus resultados. |
| **Niveles de Prueba** | Se puede realizar en el nivel de componente y de integración. |
| **Alcance** | En el nivel de prueba de integración de componentes, la prueba estructural pueden basarse en la arquitectura del sistema, como las interfaces entre componentes |
| **Cobertura** | La cobertura  estructural es la medida en que algún tipo de elemento estructural ha sido practicado mediante pruebas, y  se expresa como un porcentaje del tipo de elemento cubierto. |

Prueba asociada al cambio:

|  |  |
| --- | --- |
| **Definición** | Existen 2 tipos de prueba relacionadas al cambio:  **Prueba de confirmación**: Una vez corregido un defecto, el software se puede probar con todos los  casos de prueba que fallaron debido al defecto, que se deben volver a ejecutar en la nueva versión  de software. El  objetivo de una prueba de confirmación es confirmar que el defecto original se ha solucionado de  forma satisfactoria.  **Prueba de regresión:** Es posible que un cambio hecho en una parte del código, ya sea una  corrección u otro tipo de cambio, pueda afectar accidentalmente el comportamiento de otras partes  del código, ya sea dentro del mismo componente, en otros componentes del mismo sistema, o  incluso en otros sistemas. La prueba de regresión implica la  realización de pruebas para detectar estos efectos secundarios no deseados. |
| **Implementación** | Especialmente en los ciclos de vida de desarrollo iterativos e incrementales (por ejemplo, Agile), las nuevas  características, los cambios en las características existentes y la refactorización del código dan como  resultado cambios frecuentes en el código, lo que también requiere pruebas asociadas al cambio. |
| **Niveles de Prueba** | La prueba de confirmación y la prueba de regresión se realizan en todos los niveles de prueba. |
| **Cobertura** | Los juegos de prueba de regresión se ejecutan muchas veces y generalmente evolucionan lentamente,  por lo que la prueba de regresión es un fuerte candidato para la automatización. |





Ejemplos de tipos de prueba Los siguientes ejemplos están basados en una aplicación bancaria.

Pruebas funcionales:

*Prueba de componente:* Las pruebas se diseñan con base en la forma en que un componente debe calcular el interés a pagar por un préstamo.

*Prueba de integración de componentes:* Las pruebas se diseñan en función de cómo la información de la cuenta capturada en la interfaz de usuario se transfiere a la lógica de negocio.

*Prueba de sistema:* Las pruebas se diseñan sobre la base de cómo los titulares de cuentas pueden solicitar una línea de crédito sobre sus cuentas corrientes.

*Prueba de integración de sistemas:* Las pruebas se diseñan en función de cómo el sistema utiliza un microservicio externo para comprobar la calificación crediticia del titular de una cuenta.

*Prueba de aceptación:* Las pruebas se diseñan con base en la forma en que el empleado del banco tramita la aprobación o rechazo de una solicitud de crédito.

Pruebas estructurales:

*Prueba de componente:* pruebas están diseñadas para lograr una cobertura completa de sentencia y decisión para todos los componentes que realizan cálculos financieros.

*Prueba de integración de componentes:* las pruebas están diseñadas para evaluar cómo cada pantalla de la interfaz del navegador pasa datos a la siguiente pantalla y a la lógica de negocio.

*Prueba de sistema:* las pruebas están diseñadas para cubrir las secuencias de páginas web que pueden ocurrir durante una solicitud de línea de crédito (workflow).

*Prueba de integración de sistemas:* las pruebas están diseñadas para evaluar todos los tipos de consulta posibles que se envían al microservicio de calificación crediticia.

*Prueba de aceptación:* las pruebas están diseñadas para cubrir todas las estructuras de archivos de datos financieros soportados y rangos de valores para transferencias de banco a banco.

Pruebas asociadas al cambio:

*Prueba de componente:* se construyen pruebas de regresión automatizadas para cada componente y se incluyen dentro del marco de integración continua.

*Prueba de integración de componentes:* las pruebas están diseñadas para confirmar la corrección de defectos relacionados con la interfaz a medida que las correcciones se registran en el repositorio de código.

*Prueba de sistema:* las pruebas de un flujo de trabajo dado se ejecutan de nuevo si cambia alguna pantalla de ese flujo de trabajo.

*Prueba de integración de sistemas:* las pruebas de la aplicación que interactúa con el microservicio de calificación de crédito se vuelven a ejecutar diariamente como parte del despliegue continuo de ese microservicio.

*Prueba de aceptación:* todas las pruebas que han fallado previamente se vuelven a ejecutar después de que se haya corregido un defecto encontrado en la prueba de aceptación.

## C7A – 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.

Elección de una técnica de prueba:

* 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:*](#TecnicasDeCajaNegra) 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.

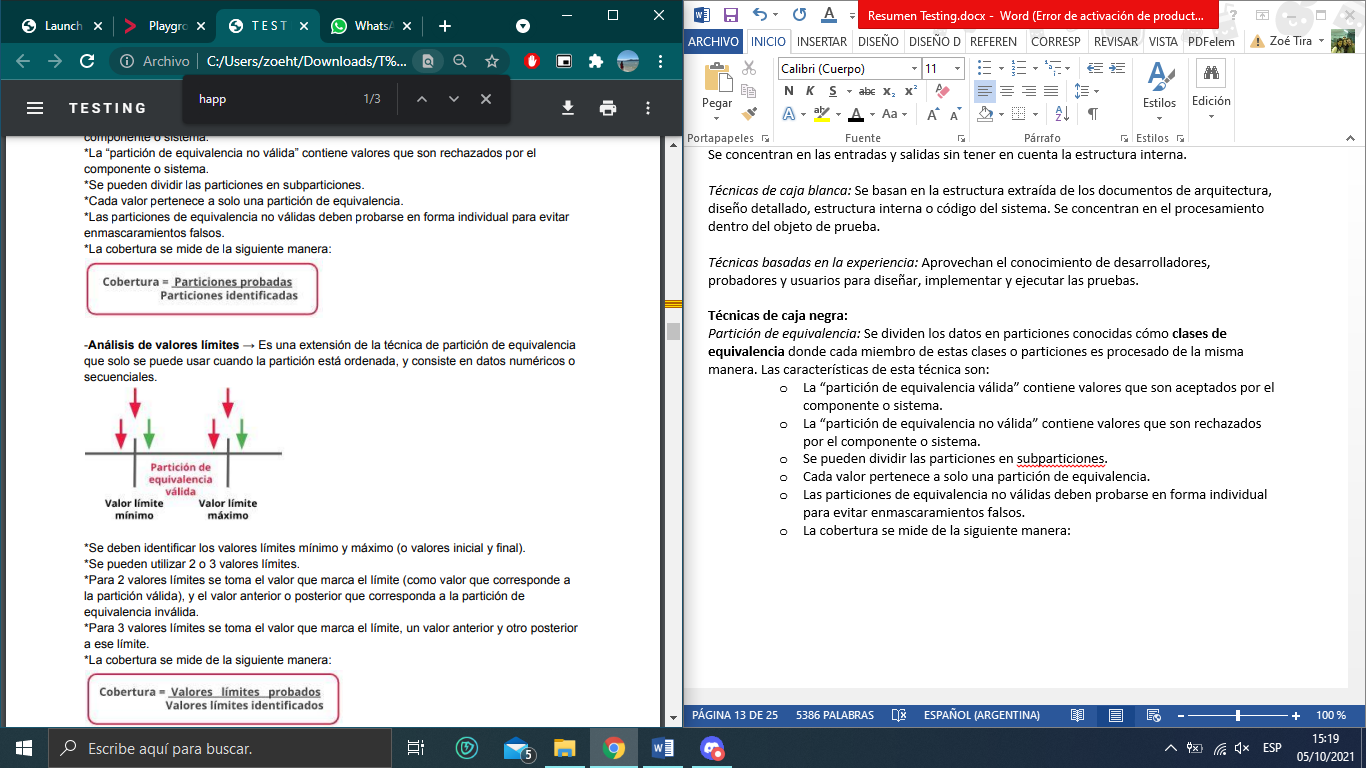
*Técnicas de caja blanca:* Se basan en la estructura extraída de los documentos de arquitectura, diseño detallado, estructura interna o código del sistema. Se concentran en el procesamiento dentro del objeto de prueba.

[*Técnicas basadas en la experiencia:*](#TécnicasBasadasEnLaExperiencia) Aprovechan el conocimiento de desarrolladores, probadores y usuarios para diseñar, implementar y ejecutar las pruebas.

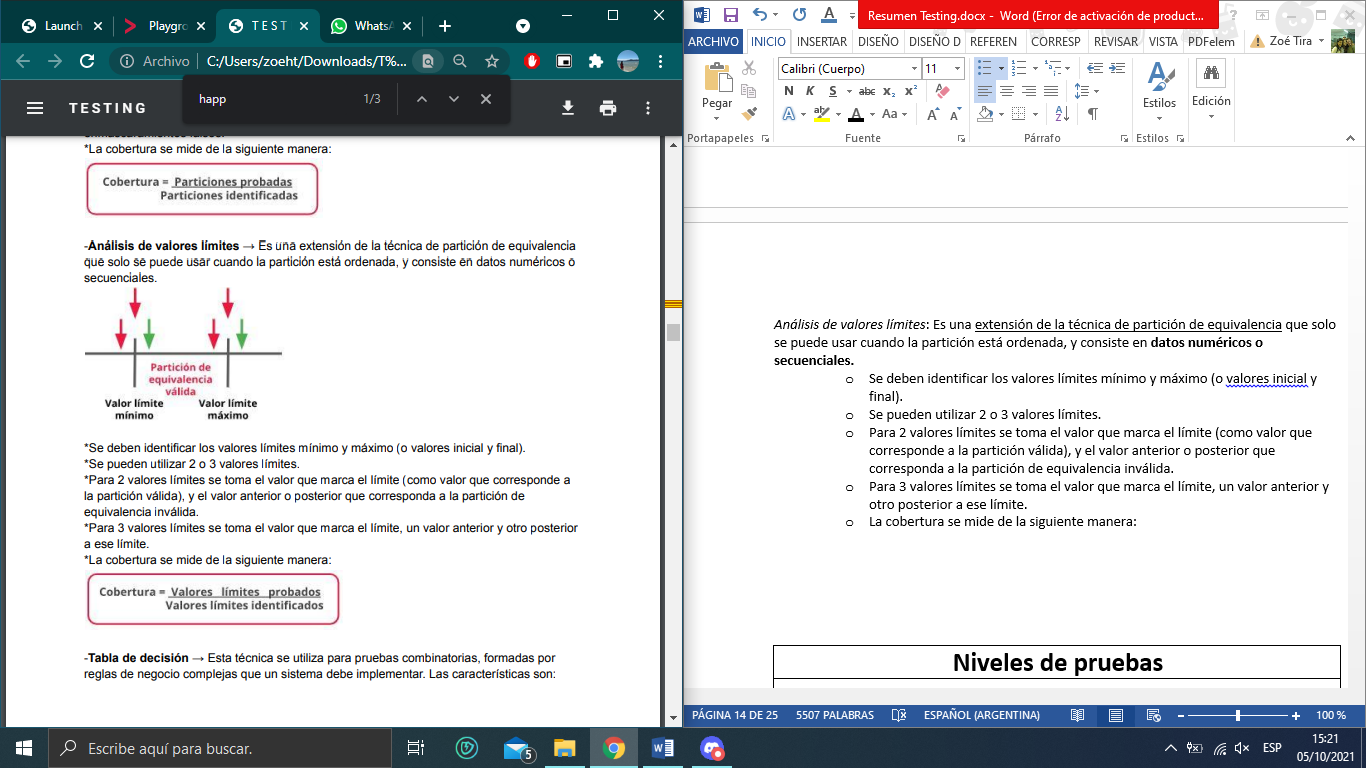
Técnicas de caja negra

*Partición de equivalencia:* Se dividen los datos en particiones conocidas cómo **clases de equivalencia** donde cada miembro de estas clases o particiones es procesado de la misma manera. Las características de esta técnica son:

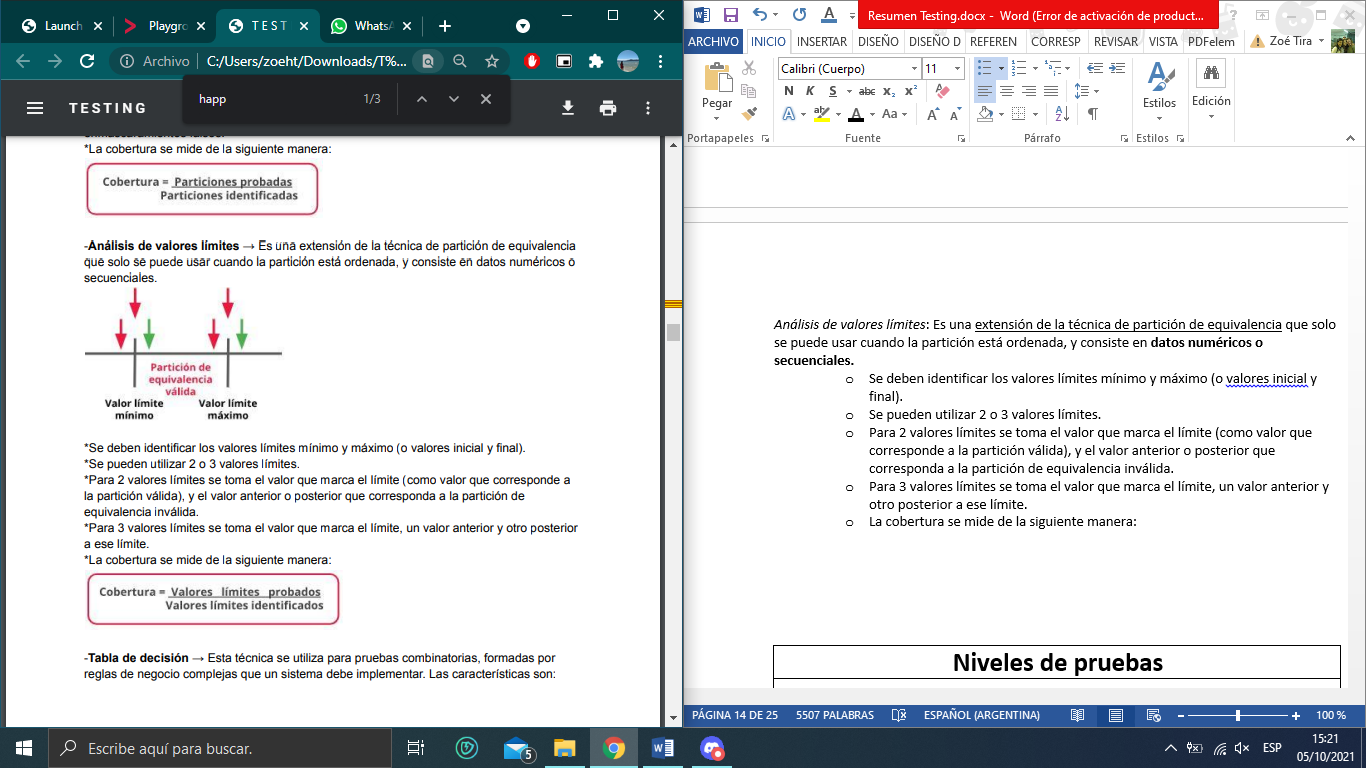
* La “partición de equivalencia válida” contiene valores que son aceptados por el componente o sistema.
* La “partición de equivalencia no válida” contiene valores que son rechazados por el componente o sistema.
* Se pueden dividir las particiones en subparticiones.
* Cada valor pertenece a solo una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



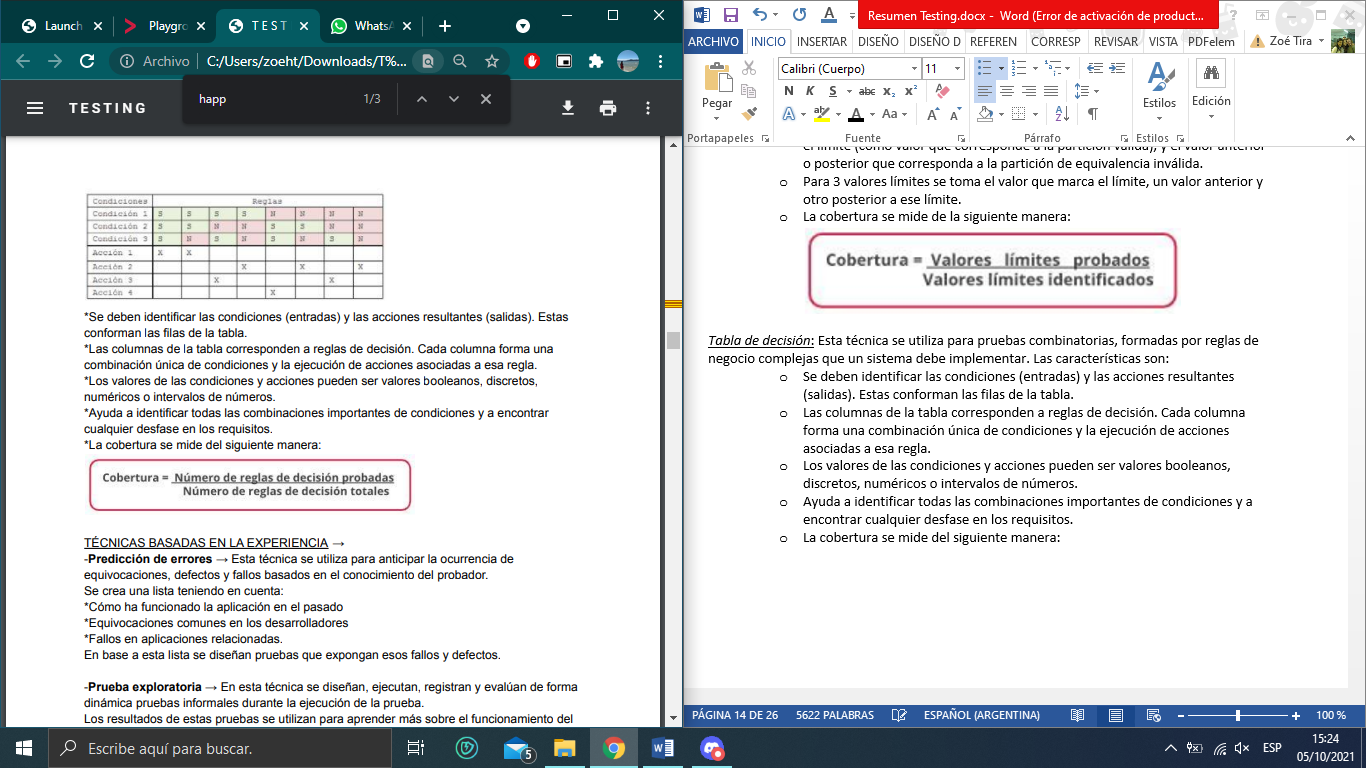
Ejemplo: colores RGB en formato **String** minúscula.

*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.**

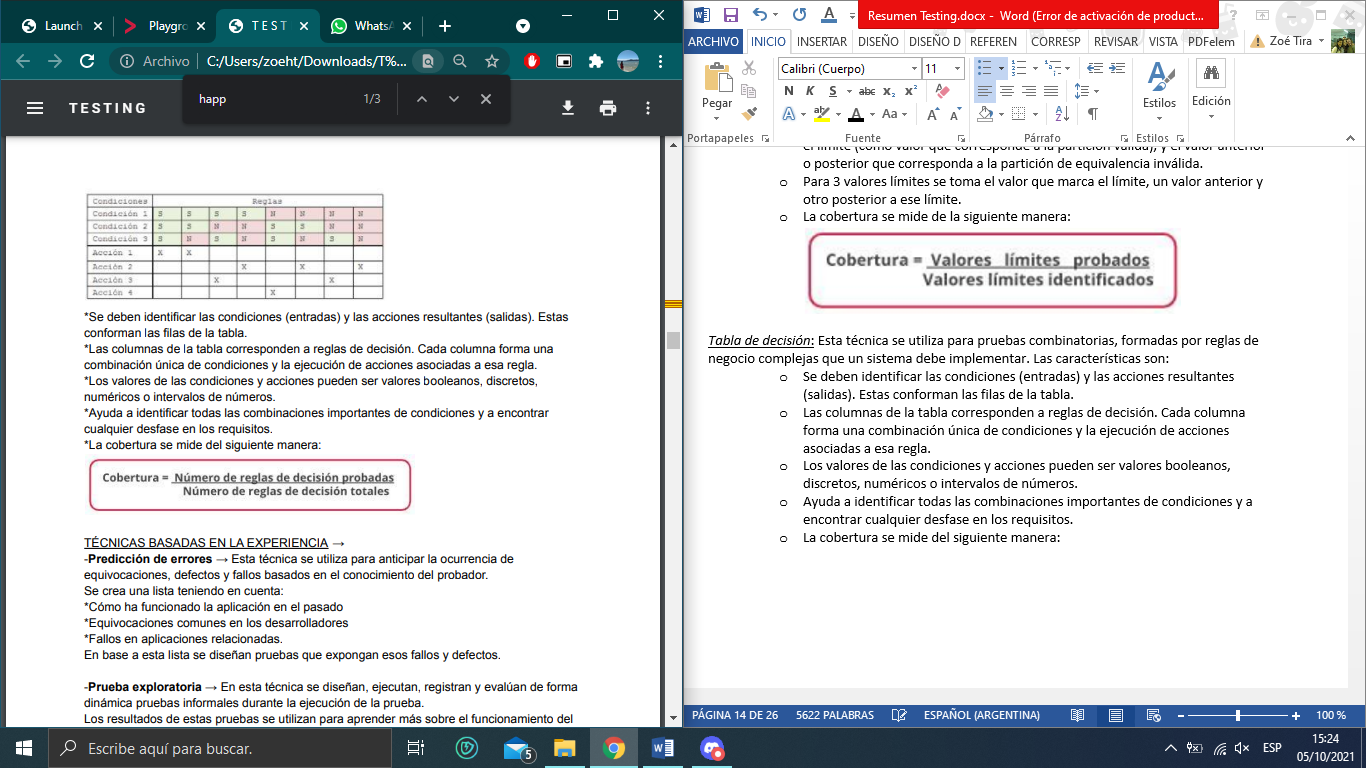
* Se deben identificar los valores límites mínimo y máximo (o valores iniciales y finales).
* Se pueden utilizar 2 o 3 valores límites.
* Para 2 valores límites se toma el valor que marca el límite (como valor que corresponde a la partición válida), y el valor anterior o posterior que corresponda a la 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:



Ejemplo: dato de entrada corresponde a un número que nos indica el día de la semana (1-7)

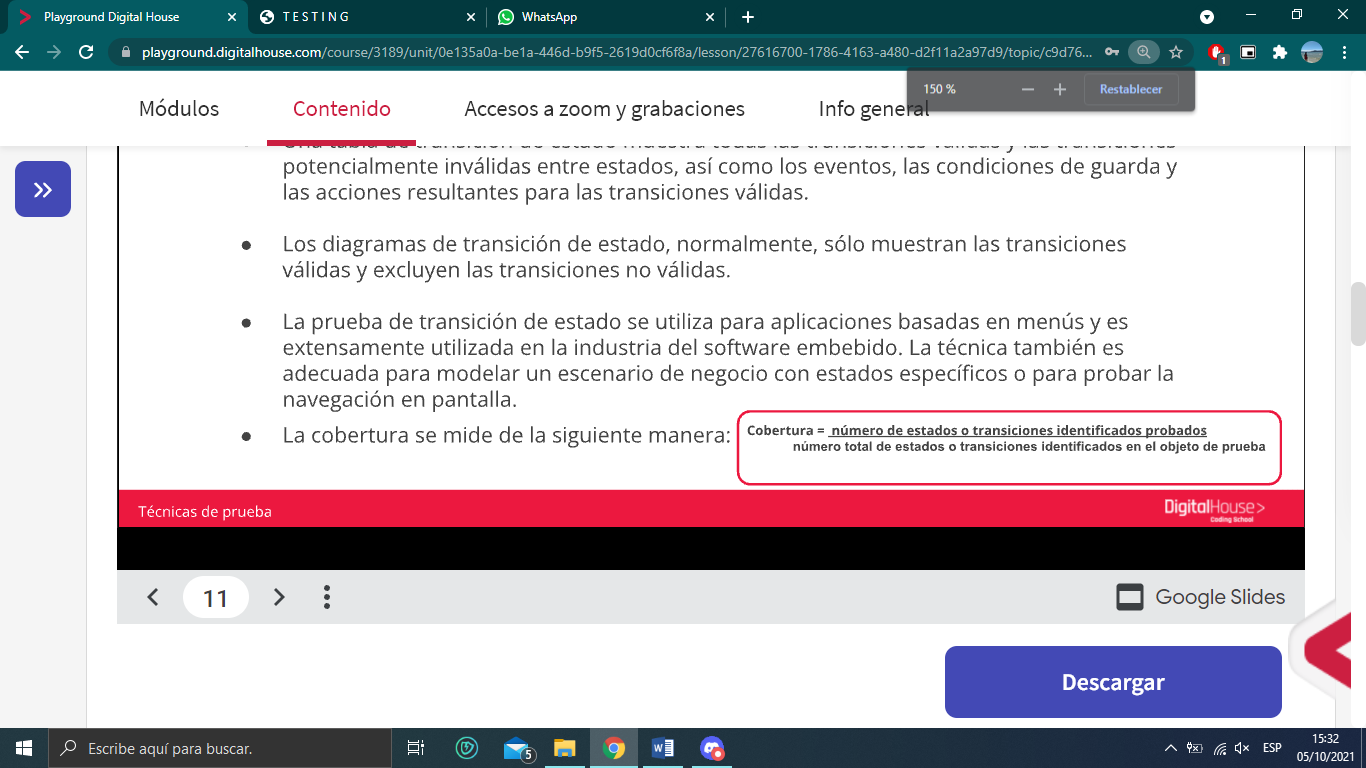
*Tabla de decisión*: Esta técnica se utiliza para pruebas combinatorias, formadas por reglas de negocio complejas que un sistema debe implementar. Las características son:

* 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 del siguiente manera:



*Transición de estados:* Muestra los posibles estados del software, así como la forma en que el software entra, sale y realiza las transiciones entre estados. Las características de esta técnica son:

* Una tabla de transición de estado muestra todas las transiciones válidas y las transiciones potencialmente invalidas entre estados, así como los eventos, las condiciones de guarda y las acciones resultantes para las transiciones válidas.
* Los diagramas de transición de estado, normalmente, sólo muestran las transiciones válidas y excluyen las transiciones no válidas.
* La prueba de transición de estado se utiliza para aplicaciones basadas en menús y es extensamente utilizada en la industria del software embebido. La técnica también es adecuada para modelar un escenario de negocio con estados específicos o para probar la navegación en pantalla.
* 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 esta 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 cómo no funcionales.

## C8A – IMPLEMENTACIÓN Y EJECUCIÓN DE LA PRUEBA

Durante la ejecución de las pruebas, los conjuntos de pruebas se ejecutan luego del despliegue de cambios en los ambientes de prueba como parte del desarrollo planificado dentro de un sprint.

La ejecución de pruebas incluye las siguientes actividades principales:

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 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 al inconveniente vemos dos conjuntos o juegos de casos de prueba muy utilizados a la hora del despliegue de nuevas funcionalidades. Son una colección de casos de prueba con un fin específico.

*Pruebas de humo:* Cubren la funcionalidad principal de un componente o sistema. El objetivo es asegurar que las funciones cruciales de un sistema funcionen, pero sin preocuparse por los detalles finos. Con pruebas sencillas y que demandan poco tiempo verificamos que funcionan correctamente ciertos caminos de la aplicación. Generalmente se eligen sólo un conjunto de funcionalidades significativas de la aplicación.

*Pruebas de regresión:* 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 éstas se ejecutan muchas veces y generalmente evolucionan lentamente. La correcta aplicación de un proceso de prueba durante el despliegue implicaría:

1. Ejecutar las pruebas de humo, y una vez confirmada la ejecución exitosa de estas pruebas estamos en condiciones de regresionar nuestro sistema bajo prueba.
2. Ejecutar pruebas de regresión.

¿Por qué es importante ejecutar estas pruebas?: 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 de la misma forma. Es probable que estas dos pruebas se complementen con pruebas específicas relacionadas a las nuevas funcionalidades que se desplegarán.

# Módulo 3 - Análisis y Planificación de la prueba

## C10A – PRUEBAS ESTÁTICAS Y DINÁMICAS

Pruebas estáticas y dinámicas

Las pruebas son una combinación de múltiples actividades del ciclo de vida del software relacionadas con la planificación, el diseño y la evaluación del producto de software, con el objetivo de encontrar los defectos y determinar si el software cumple o no con los requisitos especificados.

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. Estas se complementan entre sí y nos permiten entregar un software con la mejor calidad posible.

Prueba estática: Se basa en la evaluación manual de los productos de trabajo, es decir revisiones, o en la evaluación basada en herramientas del código u otros productos de trabajo, es decir análisis estático. Este tipo de pruebas no requieren la ejecución del software que se está probando.

Se utilizan para examinar cualquier producto de trabajo, por ejemplo:

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

Ventajas:

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

Si el defecto se encuentra en las pruebas dinámicas, solucionarlo requerirá de un cambio de código, realizar una prueba de confirmación y luego incluir los mismos en pruebas de regresión, además de los cambios de toda la documentación asociada.

*Defectos encontrados con pruebas estáticas:*

* Defectos en los requisitos (inconsistencias, ambigüedades, etc.)
* Defectos de diseño (estructura de base de dato 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.)

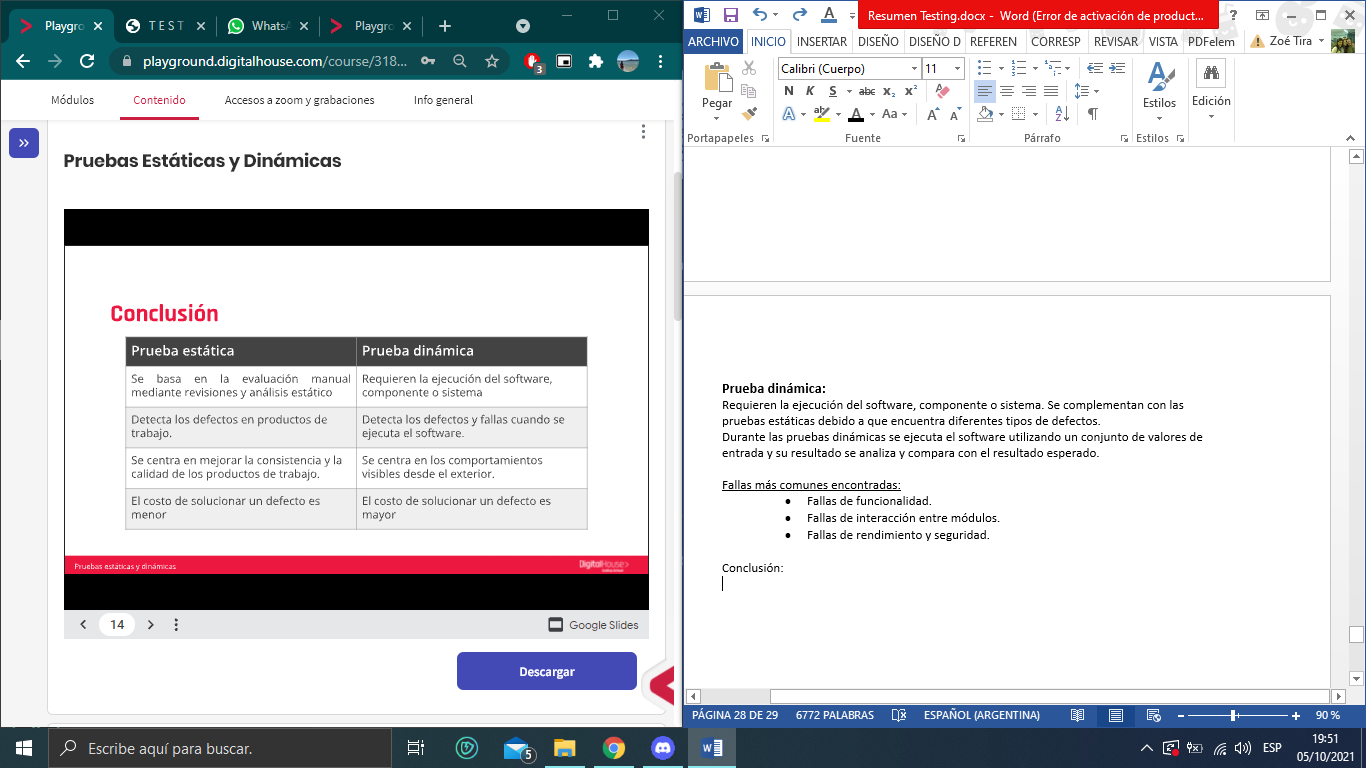
Prueba dinámica: Requieren la ejecución del software, componente o sistema. Se complementan con las pruebas estáticas debido a que encuentra diferentes tipos de defectos.

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.

*Fallas más comunes encontradas:*

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

Conclusión:



Proceso de revisión

Dentro de las pruebas estáticas, una forma de detectar errores es mediante un 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.

Componentes en el proceso de revisiones formales:

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: Asegura el funcionamiento efectivo de las reuniones de revisiones y las 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:

* *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.

Requisitos

Una de las revisiones que se realizan en las pruebas estáticas es examinar los requisitos del software. 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. Por 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).

# Módulo 4 - Análisis y Planificación de la prueba

## C11A – ORGANIZACIÓN DE LA PRUEBA

Organización de la prueba

¿Qué pasa si luego de la publicación del proyecto se debe realizar un cambio imprevisto o si se visualiza un error que necesita ser corregido? No podemos hacer los cambios directamente sobre el ambiente en el que el cliente está utilizando el software, porque podríamos romperlo y dejarlo inoperativo. Es por eso que deben existir diferentes ambientes de trabajo, donde se pueda desarrollar y probar los cambios antes de que llegue al ambiente del cliente.

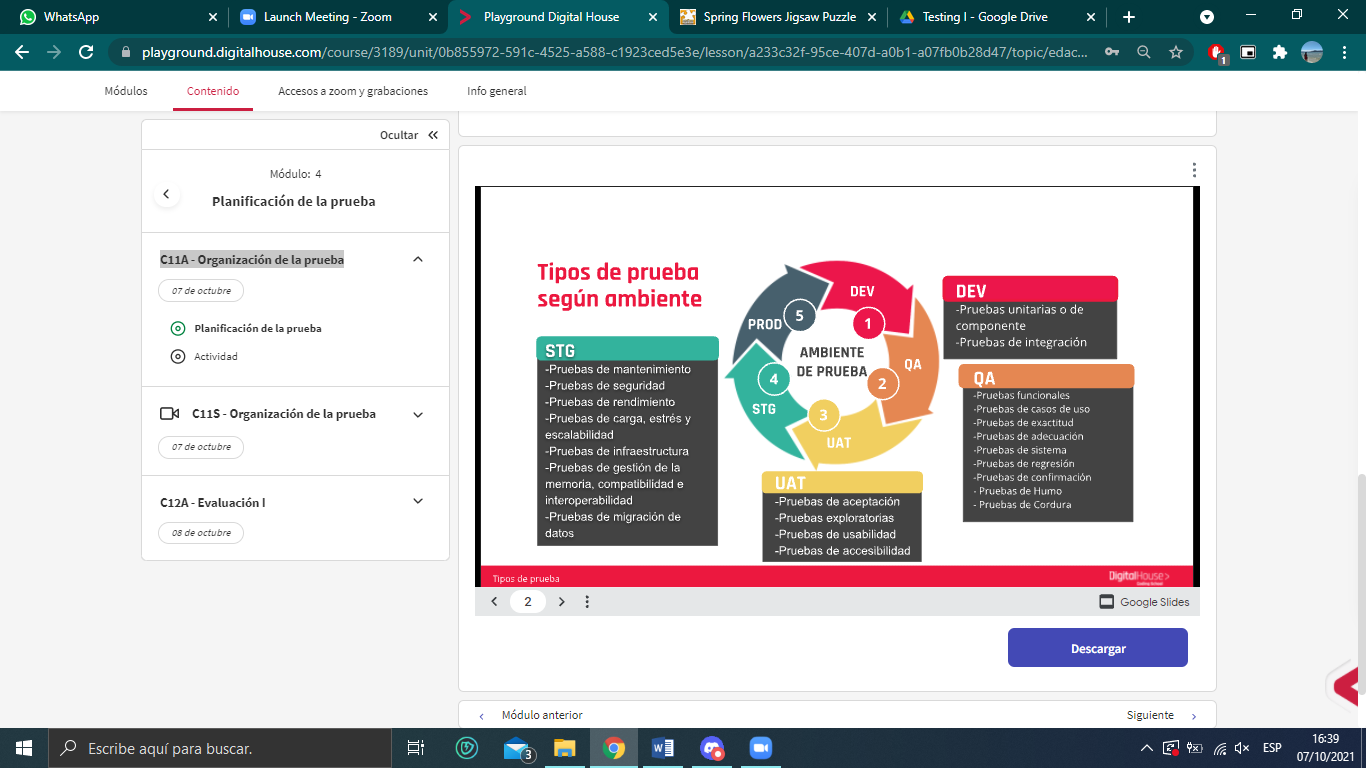
¿Qué es un ambiente?: Entorno con todos los recursos necesarios para que se pueda ejecutar un sistema. La ejecución de las pruebas se realiza en diferentes espacios de trabajo de acuerdo a la etapa del ciclo de desarrollo en el que se encuentre el sistema. Esto nos permitiría desarrollar aplicaciones de forma segura y con entornos diferenciados para realizar la programación, realizar pruebas, compartir resultados con los clientes y permitirles realizar pruebas y prácticas; y finalmente publicar una aplicación robusta y estable.

Niveles de ambientes:

Entonces, podemos decir que es conveniente distinguir los siguientes entornos:

* 1. *Ambiente de desarrollo o DEV:* este entorno es el espacio de trabajo donde el programador desarrolla el código de la aplicación, realiza pruebas iniciales y comprueba si la aplicación se ejecuta correctamente con ese código. Este ambiente puede ser local o en la nube, de acuerdo a la necesidad del proyecto.
  2. *Ambiente de pruebas o QA :* el entorno de pruebas suele estar ubicado en un servidor en la nube o en una granja de servidores locales (laboratorio). Permite minimizar incidencias en etapas posteriores, ya que el tester ejecutaría las primeras pruebas de funcionalidad en este ambiente.
  3. *Ambiente de UAT:* el entorno de UAT (o de pruebas de aceptación de usuario) permite a los usuarios del cliente poder verificar que los cambios realizados son los que realmente se solicitaron, evaluando a su vez accesibilidad y usabilidad.
  4. *Ambiente de preproducción o STAGE:* este entorno debería poseer una configuración técnica idéntica a la que nos encontraremos en el entorno de producción. El propósito principal de este entorno es emular al entorno de producción con el fin de probar las actualizaciones y asegurar que estas no corromperán la aplicación en los servidores en producción cuando sean desplegadas. De esta forma se minimizan las caídas del sistema y corte de los servicios en producción.
  5. *Ambiente de producción o PROD:* este es el entorno donde finalmente se ejecuta la aplicación, donde acceden los usuarios finales y donde se trabaja con los datos reales de negocio. Es un servidor que posee las mismas características y configuración que tendrá el servidor de preproducción. Aunque, en este caso, puede estar configurado por más de un servidor, para efectos de balanceo de carga en aplicaciones que requieren una infraestructura con capacidad de manejar un tráfico de usuarios pesado y miles de conexiones concurrentes.

¿Qué tipos de pruebas se pueden hacer en cada ambiente?



En el ambiente de PROD (producción) no se desarrollara ningún tipo de prueba ya que los probadores no tienen acceso a este ambiente y, en caso de que tengan acceso, no se deben realizar acciones que generen datos, se corre el riesgo de ingresar dato basura, se interfiere en los datos de seguimiento (creo que el profe dijo que podía ser catastrófico).

DEV:

* *Pruebas unitarias o de componente*: Se centra en los componentes que se pueden probar por separado. Objetivo: encontrar defectos en el componente y verificar que los comportamientos funcionales y no funcionales del componente son los diseñados y especificados.
* Módulo 4 – Unit Testing

## C13A – INTRODUCCIÓN AL DEBUGGING

Introducción a la prueba de componente

Debugging: Proceso utilizado por los programadores para encontrar, analizar y remover la causa de los fallos en el software, errores en el código fuente.

Se puede realizar el debug de una aplicación utilizando:

* Las herramientas del desarrollador desde la consola del navegador, ej: Chrome Dev Tools, Firefox Dev Tools
* La opción Debug dentro del framework o IDE utilizado para el desarrollo, ej: Visual Code, Visual Studio.

Estrategias de Debugging:

*Depuración por fuerza bruta:* Es la más común pero la menos eficiente. Se invocan señales en tiempos de ejecución y se carga el programa con instrucciones de salida, entonces en algún lugar del código donde se produce el error se puede encontrar una pista que pueda conducir a la causa del error. (Básicamente meter console.log con mensajes por todos lados a ver que fallo).

*Backtracking:* Enfoque de depuración muy común en pequeños programas que inicia donde se encontró un síntoma y se recorre hacia atrás el código fuente hasta encontrar la causa del problema.

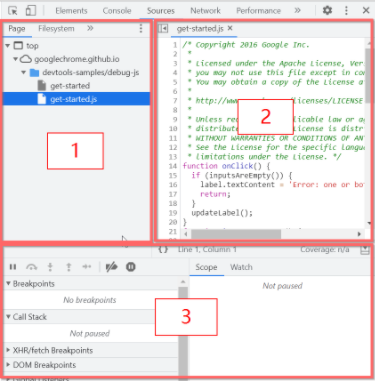
*Eliminación de causas:* Los datos relacionados con el error se organizan para aislar la causa posible elaborando una hipótesis de la causa.

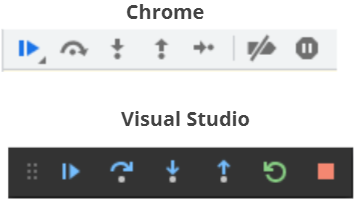
Una particularidad compartida entre todas estas estrategias es el enfoque, ya que se complementan con las herramientas de depuración que proporcionan soporte semi-automatizado al desarrollador.

IDEs: proporcionan la manera de capturar algunos errores típicos y específicos del lenguaje como caracteres de final de línea, variables indefinidas, sin requerir compilación. Cuentan con compiladores de depuración, ayudas dinámicas para la depuración, generadores automáticos de casos de pruebas y herramientas de corrección de referencias cruzadas.

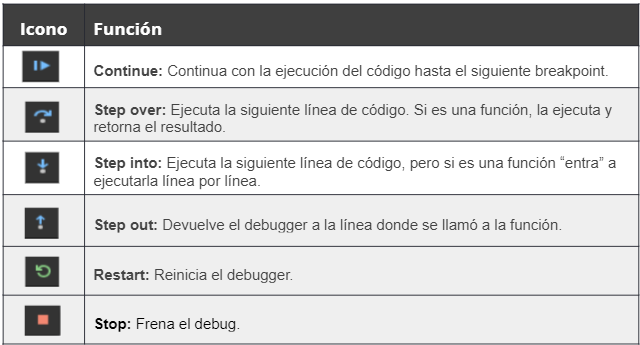
Breakpoints: Punto de interrupción en nuestro código para detener la ejecución del programa en líneas específicas y analizar la situación del mismo, revisando por ejemplo el estado de las variables o de la pila de llamadas en ese momento.

Debug desde la consola de Chrome: Se puede depurar código JavaScript directamente desde la consola de Chrome, siguiendo los pasos a continuación:

1. Acceder a la consola de Chrome.
2. Ir a la pestaña “Sources”, que cuenta con 3 partes:
3. Navegador de archivos
4. Editor de código
5. El depurador de Javascript
6. Marcar el breakpoint en la línea de código correspondiente, haciendo clic en el número de la misma.
7. Comenzando con alguna función se inicializa el modo debug, por ejemplo un click en un botón, luego se puede presionar F11 para recorrer linea por linea y F8 para recorrer de un breakpoint a otro.

También, se pueden utilizar los comandos de la propia interfaz que se verán a continuación:

* Continue/Resume
* Step over
* Step into
* Step out
* Restart
* Stop



Debug en Visual Studio:

1. Dentro de la aplicación, Ir a la funcionalidad que se requiere depurar, desde el explorador de soluciones.
2. Colocar un breakpoint en la línea de código requerida.



1. Iniciar el Debug, haciendo clic en F5 o ingresando en Ejecutar(Run) -> Start Debugging.

Si es la primera vez que se va a realizar debugging sobre la aplicación, se debe ingresar al menú “Add configuration…” y agregar la configuración de Chrome para ejecutar la aplicación. Esto crea el archivo launch.json en el cual se debe configurar en “file” la dirección del archivo .html que inicia la aplicación.

1. Debido a que se utiliza Chrome para realizar el debugging, es necesario realizar una acción en la aplicación.

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 las fallas del código. |
| Es la investigación y detección del error. | Es la visualización de errores. |
| Realizado generalmente por el programador o desarrollador, a excepción del código generado por el tester como parte de los scripts de prueba automáticos. | Realizado generalmente por el tester. |
| No se puede realizar sin conocimiento de diseño adecuado. | No es necesario tener conocimiento 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 de 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 componentes, pruebas de integración, pruebas de sistema, pruebas de aceptación. |
| No es una etapa del ciclo de vida del desarrollo del software, ocurre como consecuencia de las pruebas. | Es una 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 antes de tener el código (testing estático) o después de que se escribe el código (testing dinámico). |

## C14A – INTRODUCCIÓN A LA PRUEBA DE COMPONENTE

Prueba de componente o prueba unitaria (unit test)

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).

Unidad: puede ser casi cualquier parte del código que queremos que sea. 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.

Introducción a unit testing / prueba de componente

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

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

Proceso: 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.
4. Si el test pasa, se confirma el resultado esperado.
5. 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 el costo de las pruebas, ya que los defectos se capturan en una fase temprana.
* Mejora el diseño y código del software debido a que permite una mejor refactorización del mismo.
* 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:* Herramienta que ejecuta los test y muestra los resultados en forma de reporte. Por ejemplo: Mocha (<https://mochajs.org/>)
* *Assertion Library:* Herramienta que se utiliza para validar la lógica de prueba, las condiciones y resultados esperados. Por ejemplo: Chai (<https://www.chaijs.com/guide/>)

JEST es un framework que incluye tanto el test runner como la assertion library.

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:* Marco de trabajo de pruebas unitarias ampliamente utilizado para todos los lenguajes .net. Es de código abierto y admite pruebas basadas en datos que pueden ejecutarse en paralelo.
* *JMockit:* Herramienta de prueba unitaria de código abierto. Es 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:* 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:* 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.

Instalando el framework Jest: Para comenzar a crear nuestros casos de prueba unitarios lo que vamos a necesitar es tener instalado las siguientes herramientas:

* NodeJS: Es un entorno de ejecución para JS construido bajo el motor V8 de google Chrome, por lo que permite ejecutar JS fuera de un navegador.
* IDE: Visual Studio Code.
* Jest: VIDEO DE 23 MIN JAJAJ

## C16A – Prueba de componente

Prueba de componente

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. 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 componentes.

*Prueba y cobertura de sentencia:* Nos referimos al porcentaje de sentencias ejecutables que han sido practicadas por un juego de pruebas. Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute (al menos) una vez.

* 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: Es aquella prueba en la que se escriben test cases suficientes para que cada decisión en el programa se ejecute una vez con resultado verdadero y otra con el falso.

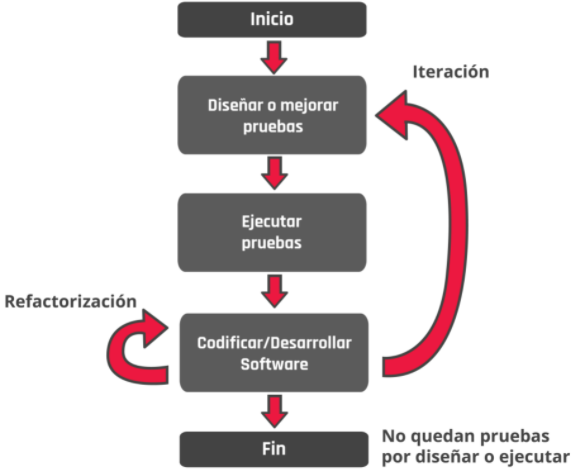
* Ejercita las decisiones en el código y prueba el código que se ejecuta basado en los resultados de la decisión.
* Los casos de prueba siguen los flujos de control que se producen desde un punto de decisión.
* En el caso de un IF se necesitan dos casos de prueba como mínimo, uno para el valor VERDADERO 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: Práctica de desarrollo de software que consiste en escribir las pruebas unitarias, luego el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito. Este enfoque de “prueba primero”, colabora en mitigar los cuellos de botella críticos que obstruyen la calidad y la entrega del software. Con base en la retroalimentación, corrección de errores y 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 insumir tiempo adicional recreando extensos scripts y suites 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. Contrariamente, TDD propone que lo primero que se debe hacer es escribir los tests y luego codificar el software.

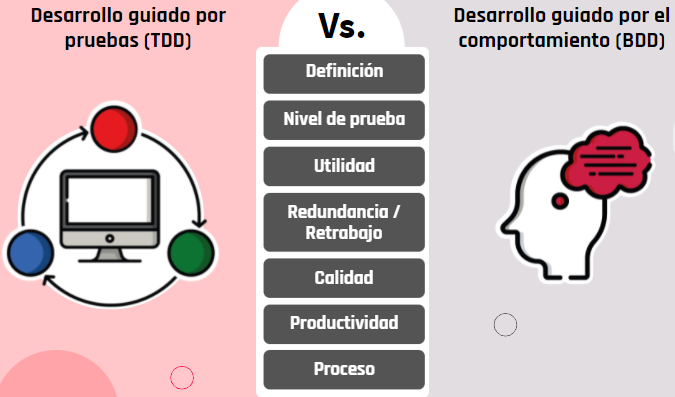
Test Driven Development:



TDD propone pensar y comprender primero el problema en su totalidad, antes de plantear la solución. Para ello, podemos encontrar los siguientes beneficios en su uso y aplicación:

* Se gana visibilidad al redactar primero los criterios sobre la totalidad del problema a solucionar. Luego procedemos a escribir el código.
* Facilita la tarea de resolver un problema a la vez (como plantean los marcos ágiles).
* Permite iterar una vez que tenemos un código base funcional.
* Libera la “presión” de escribir un código prolijo y performante al primer intento, dado que se prioriza el funcionamiento adecuado y, luego, nos enfocamos en las mejoras que puedan aplicarse.
* Ayuda a trabajar en la precisión del código necesario (ni más, ni menos que lo que se requiere).

TDD vs. BDD:



|  |  |  |
| --- | --- | --- |
| **Desarrollo guiado por pruebas (TDD)** |  | **Desarrollo guiado por el comportamiento (BDD)** |
| Proceso de desarrollo de software donde se desarrolla el código guiado por casos de prueba automatizados. | Definición | Proceso de desarrollo de software que permite al desarrollador concentrarse en probar el código basándose en el comportamiento esperado del software. |
| Nivel unitario, se centran en el código, aunque también pueden escribirse pruebas a nivel de integración o de sistema. | Nivel de prueba | Nivel de sistema e integración, aunque también se pueden utilizar para escribir pruebas unitaria. |
| 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. |
| 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. |
| 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. | Proceso | 1. Se busca un lenguaje común 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 user story, puede utilizarse el marco de desarrollo guiado por el comportamiento para definir criterios de aceptación basados en el formato dado - cuando - entonces.  3. Se escribe el código del software de acuerdo a los criterios de aceptación estructurados.  4. Se genera el código para los casos de prueba.  5. Se ejecutan los casos de prueba y se refactoriza. |

## C17A – Test unitario

Test unitario

Nuestro primer test unitario con JavaScript:

Configurar el Framework: Para JavaScript vamos a utilizar el framework Jest. Si queremos configurarlo, debemos instalar:

1. Un IDE (el recomendado es Visual Studio Code (https://code.visualstudio.com)
2. Node.js (<https://nodejs.org>).

Debemos tener en cuenta que si el código brindado no posee el archivo **package.json** debemos crearlo, ya que aquí se van a guardar todas las configuraciones de nuestro proyecto. Para crearlo debemos correr en la terminal el comando: **npm init –y**

1. JEST (https://jestjs.io/)

Para incluir Jest dentro de nuestro proyecto debemos correr en la terminal el comando: npm install --save-dev jest (tener en cuenta que esto debemos realizarlo en cada uno de los proyectos que vamos a trabajar).

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.

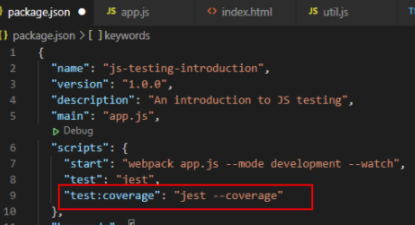
En Jest, el reporte de cobertura de prueba viene integrado con el framework.

Simplemente debemos escribir el comando npm run **test:coverage** en la terminal para ejecutar los tests. Pero antes es necesario realizar algunas configuraciones en nuestro proyecto.

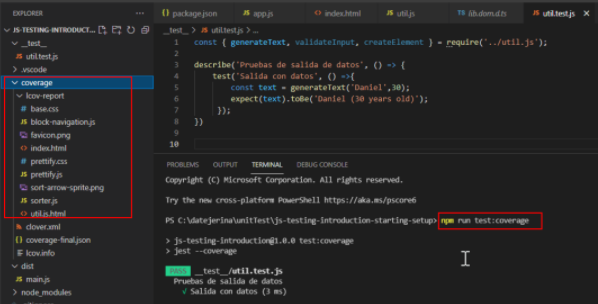
Veamos cómo es el paso a paso para obtener el reporte de cobertura y cómo podemos mejorar la cobertura de:

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

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). Este se agrega en el nodo “scripts”, como se muestra a continuación:



1. Ejecutar el reporte de cobertura: Esto debemos hacerlo 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 qué 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.

1. Revisar el reporte de cobertura: En el mismo figuran 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.

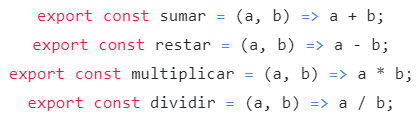


1. Agregar más casos de prueba: En caso de que sea necesario alcanzar una mayor cobertura, se pueden agregar más casos de prueba y luego ejecutar nuevamente el comando npm run test:coverage desde la terminal para regenerar el reporte.

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

Matchers más usados con Jest

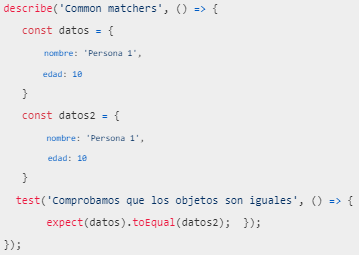
Jest usa los matchers para probar los diferentes valores que puede tener el código.



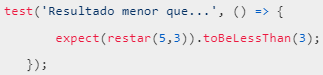
*.toBe:* Usado para comparar valores primitivos (enteros, flotantes, etc.).



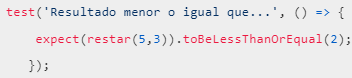
*.toEqual:* Usado para comparar objetos y todas sus propiedades:



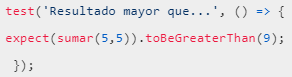
*.toBeLessThan:* El valor es menor que.



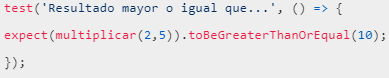
*.toBeLessThanOrEqual*: El valor es menor o igual que.



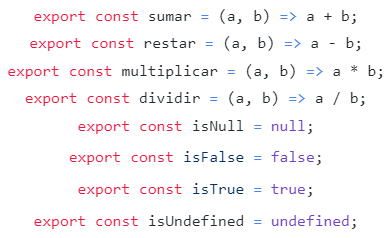
*.toBeGreaterThan:* El valor es mayor que.



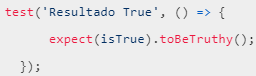
.*toBeGreaterThanOrEqual:* El valor es mayor o igual que.



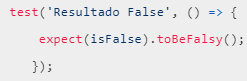
Agregaremos otras operaciones de lógica, las cuales nos permitirán comparar valores booleanos, indefinidos y nulos.



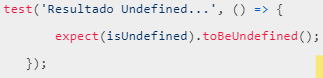
*.toBeTruthy:* El valor es verdadero.



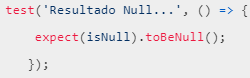
*.toBeFalsy*: El valor es falso.



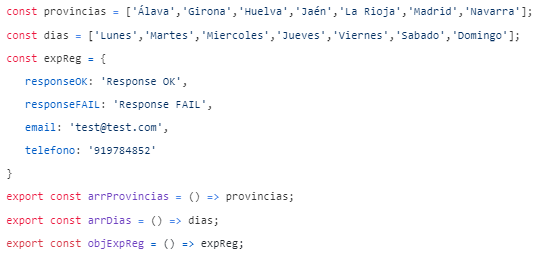
*.toBeUndefined:* El valor es undefined.



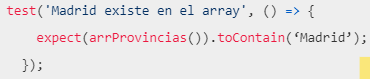
*.toBeNull:* El valor en null.



Arrays y strings:



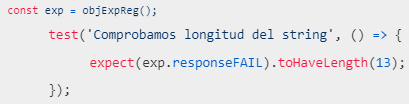
*.toBeContain:* Contiene el elemento dentro del array.



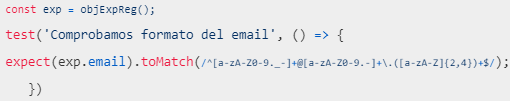
*.toHaveLength (array):* El array tiene la longitud.



*.toHaveLength (string):* También podemos usar este matcher para ver la longitud de un string.



*.toMatch:* Comprueba que un texto coincida con una expresión regular.



# Módulo 6 – Be testing

## C19A – Introducción a 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 front end, 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.

API – Application Programming Interface: Las APIs son desarrolladas para que dos sistemas puedan comunicarse, por eso devuelven información.

*Endpoint:* Punto de conexión donde necesitamos apuntar para obtener la información que queremos, son las URL.

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.

GET: Método a través del cual vamos a poder solicitar información al servidor. Solo deben recuperar datos, no modificar información. Solamente las rutas que usen el método GET podrán ser accedidas desde la URL o desde un enlace. Los demás métodos necesitaran ser procesados.

POST: Método a través del cual vamos a enviar información al servidor. Las peticiones hechas por POST servirán para generar un registro dentro de una aplicación. Es un método mucho más seguro para enviar información que GET, ya que con GET toda la información que se envíe será visible en la URL del navegador. Por el contrario, cuando enviemos información a través de POST la información esta oculta. Estos medios no pueden ser cacheables, no pueden estar en favoritos ni en el historial del navegador.

PUT: Se usa para reemplazar información en un registro que ya se encuentra en el servidor.

DELETE: Borrará los datos que están en el servidor. Se necesitara el id de lo que queremos borrar.

VIDEO DE POSTMAN: Ver resumen de front II, te enseña a usar Postman.

## C20A– API Testing

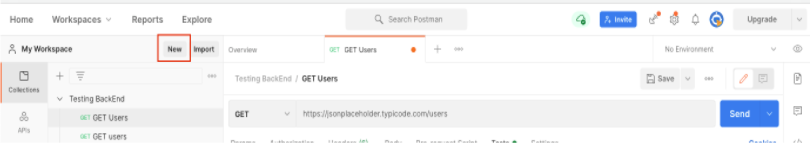
API Testing - Métodos GET Y POST

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

En el ejemplo analizaremos la obtención de usuarios desde un API externa.

*Pasos:*

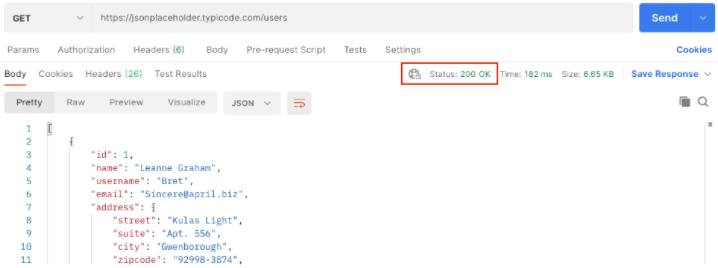
1. Primero, debemos crear una nueva solicitud en Postman. Para realizarlo, se debe hacer clic en la pestaña ‘’New’’.



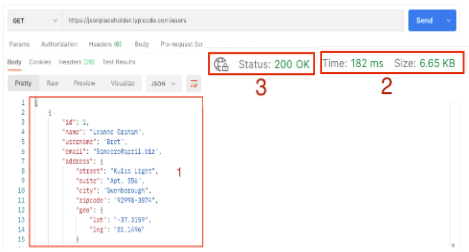
1. El siguiente paso es crear la solicitud.
2. Configura su solicitud HTTP en GET.
3. Ingresa el enlace en la URL de la solicitud (<https://jsonplaceholder.typicode.com/users>)
4. Haz clic en ENVIAR para mandar la solicitud al servidor que aloja la URL.



1. Cuando aparezca el mensaje 200 OK significa que se la solicitud se realizó correctamente.



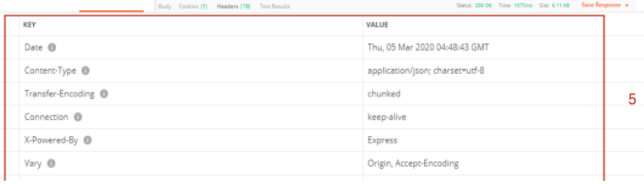
Resultados:

1. *Response:* es la información en plano devuelta por el servidor. Con esto podemos dar una revisión temprana de los datos de la aplicación.
2. *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.
3. *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 200 nos indica que la solicitud se realizó con éxito.
4. Cookies: nos permiten ver la información relacionada con la sesión.

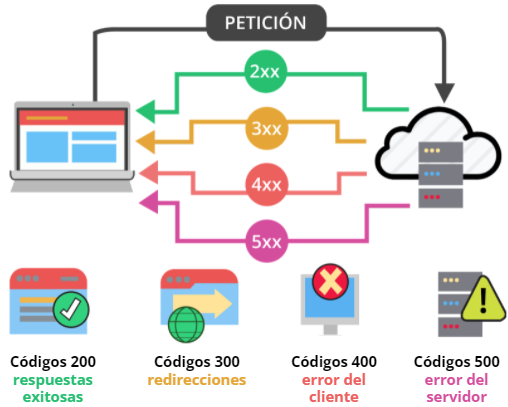
(Pestaña Headers [En el medio de la app] -> Cookies [En la parte de debajo de la app])



1. Headers: información sobre la solicitud procesada. (Al lado de Cookies).



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

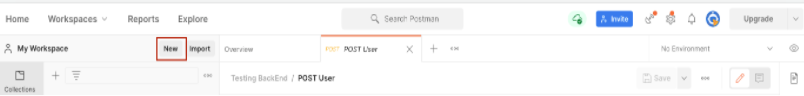


Link a códigos de estado: <https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP>

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. En el ejemplo veremos la creación de un usuario y la respuesta del API.

*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’’.

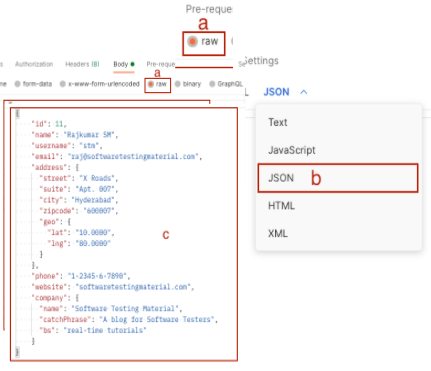


1. El siguiente caso es crear la solicitud:
2. Configurar su solicitud HTTP en POST.
3. Ingresar el enlace en la URL de la solicitud (<https://jsonplaceholder.typicode.com/users>).



1. 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:

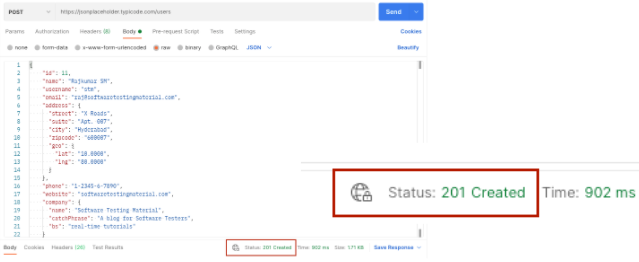
* *Raw:* Se envía la información como una cadena tipo texto, a través de un archivo tipo JSON.
* *x-www-form-unlencoded:* Se envían los datos como si fuera un formulario.



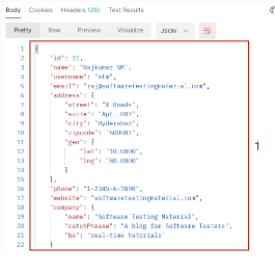
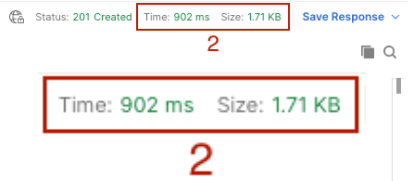
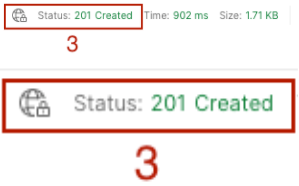
1. En este ejemplo enviaremos los datos en formato Raw. Para ello haz clic en el cuerpo de la solicitud y selecciona la opción “raw” (a), luego “Json” (b). Finalmente copia y pega el ejemplo brindado en la diapositiva anterior en el body (c).
2. Haz clic en ENVIAR para mandar la solicitud al servidor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



1. Si aparece el mensaje 201 CREATED, significa que se la solicitud se realizó correctamente.



Resultados:

1. *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.
2. *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.
3. *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.

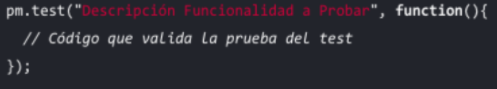
Automatización de API’s con Postman Tests y JavaScript: Postman Tests permite asegurarnos de que una API funciona como se esperaba. Nos permite establecer que las integraciones entre los servicios funcionen de manera confiable y verificar que los nuevos desarrollos no hayan roto ninguna funcionalidad existente. Nos ayuda a verificar resultados, como el estado exitoso o fallido, la comparación de los resultados esperados, etc.

Esta herramienta nos brinda un conjunto de fragmentos de código JS con algunas pruebas por defecto, pero también somos libres de escribir nuestras propias pruebas utilizando JS.

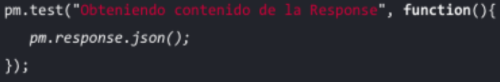
*Pasos:*

1. Comencemos con algunos conceptos a tener en cuenta:

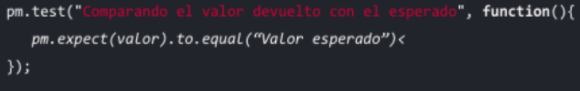
* Para codificar los test con Postman debemos de conocer un poco el API que nos ofrecen. Cada uno de los test es ejecutado con el objeto pm y en concreto con el método .test(). Así, para cada uno, tendremos la siguiente estructura:



* 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.

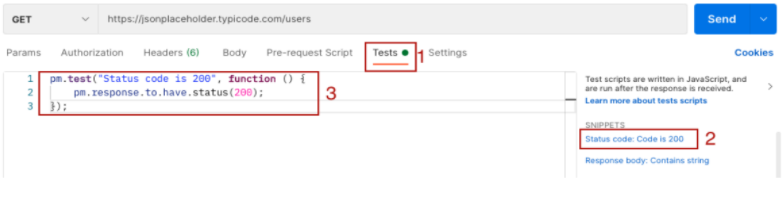


* Otro método importante es el que nos permite realizar una comprobación de contenido, este es pm.expect.

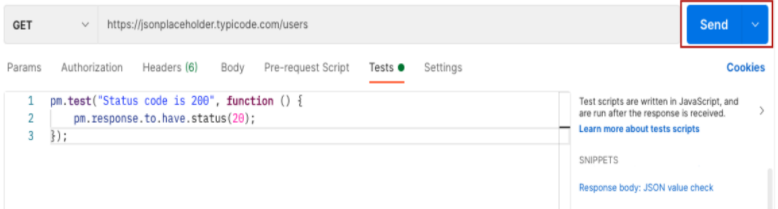


1. Teniendo en cuenta los conceptos ya definidos, veremos dos de las pruebas más utilizadas en el Testing de APIs. Postman nos brinda una serie de fragmentos por defecto que nos guía a la hora de construir nuestras pruebas:

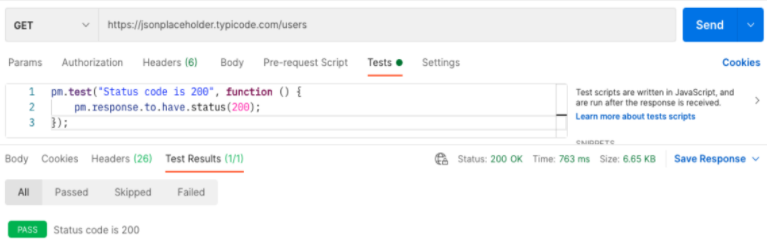
* En la solicitud GET que creamos anteriormente seleccionamos la pestaña Pruebas (1). En esta sección escribiremos nuestro set de pruebas relacionados con esa API. En la subsección de fragmentos, haremos clic en "Código de estado: el código es 200" (2) para generar una de las pruebas por defecto. La secuencia de comandos se completará automáticamente (3).



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

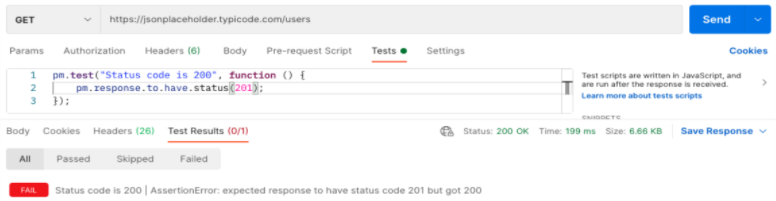


* Con esta prueba estamos validando que el código de respuesta de la API sea 200. Si esto es correcto el test devolverá PASS: eso significa que el servicio está respondiendo según lo esperado.



* 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 JS.

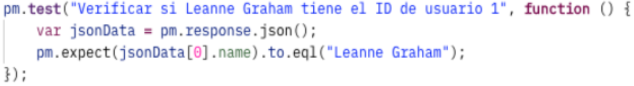


1. Agreguemos otra de las pruebas más usadas. En esta compararemos el resultado esperado con el resultado real.

* 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).



* 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.



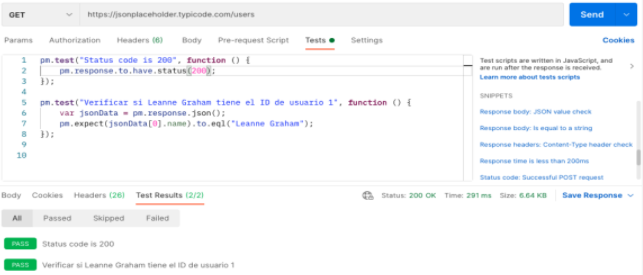
* Al hacer clic en ENVIAR se mostrará el resultado.



* 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.



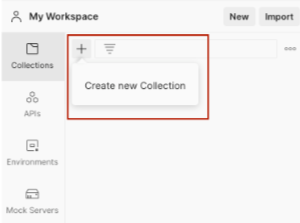
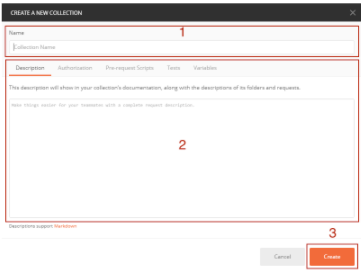
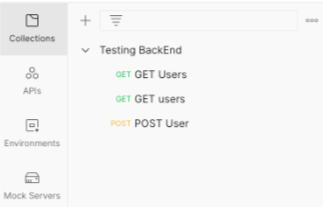
* Por último, se observa que al momento de enviar la petición se ejecutan todos los test relacionados a esta. De esta manera, podemos crear un set de test vinculados a cada petición y verificar rápidamente su estado.



API Testing - Colecciones y variables de entorno con Postman

Colecciones: 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. ¿Cómo?
3. Ingresa el nombre de la colección.
4. Ingresa una descripción para la misma (opcional).
5. Haz clic en CREAR para crear la nueva colección.
6. Puedes agregar cualquier número de peticiones a una colección. Solo debes arrastrar la petición a la carpeta de la colección.

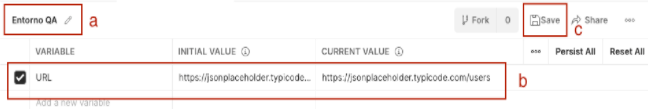
Variables de entorno: Solemos utilizar la misma solicitud varias veces con datos diferentes. Postman nos permite parametrizar estos datos y guardarlos en forma de archivo o en variables de entorno. Una variable de entorno se guarda en el entorno de trabajo. Estas se pueden crear de manera estática o dinámica. Podríamos tener diferentes entornos para Dev, QA y Producción, con sus respectivas variables.

Pasos para crear una variable de entorno:

1. Hacer clic en el botón del ojo para crear nuestras variables de entorno.



1. El siguiente paso es completar los datos del entorno y de las variables.
2. Ingresa el nombre del entorno (DEV, QA o Producción).
3. Ingresa las variables de ese entorno. Para ello hay que completar el nombre de la variable y su valor.
4. Finalmente haz clic en el botón Guardar para efectuar los cambios.



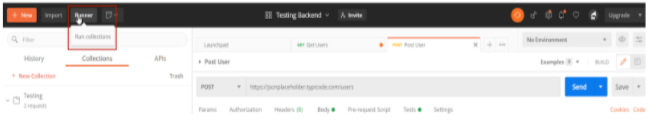
1. Por último, se crean mediante el uso de llaves dobles con el nombre de la variable a utilizar.

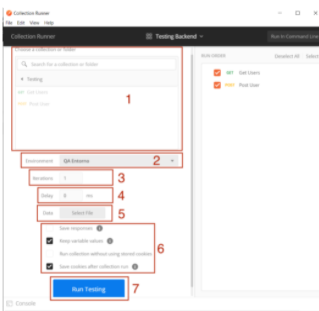


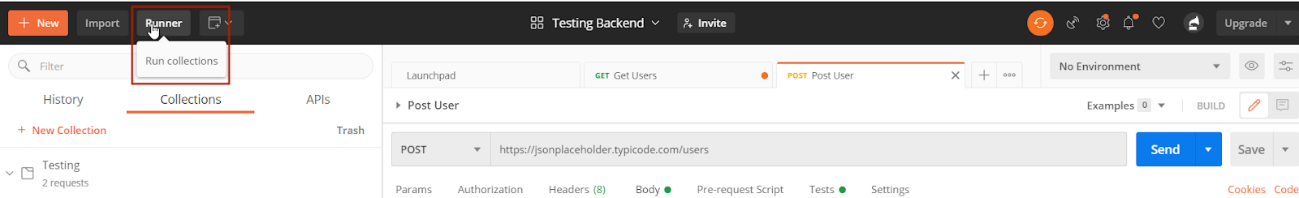
Runner para correr colecciones: El runner nos permite ejecutar un conjunto de test de diferentes colecciones al mismo tiempo, otorgando un informe de resultados.

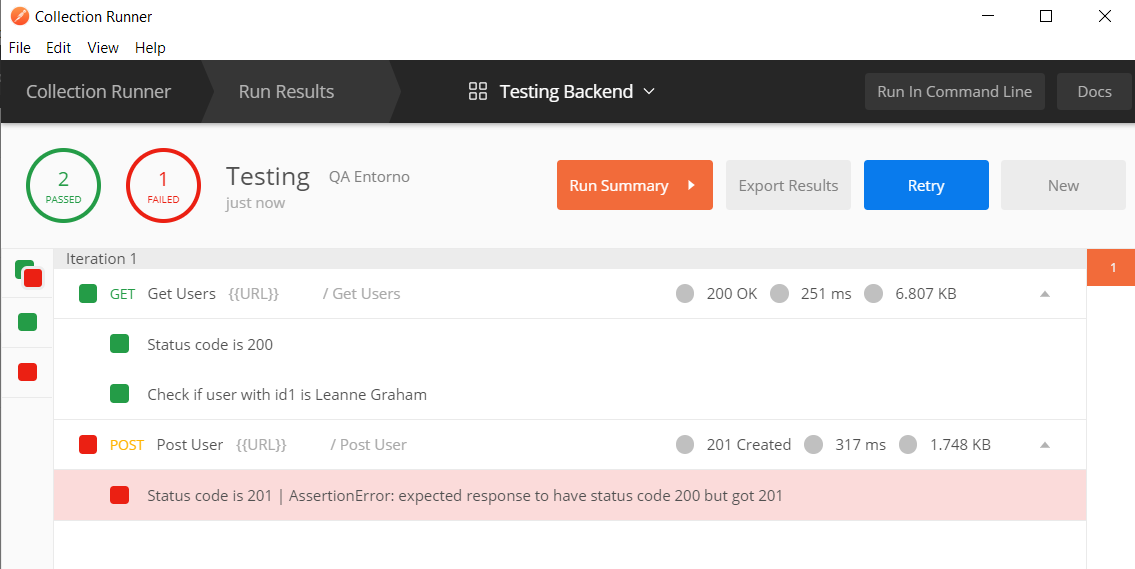
*Pasos para ejecutar el runner:*

1. Primero, se debe hacer clic en el botón Runner.



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





# Módulo 7 - Introducción a Automation

## C22A - Fundamentos de automatización de la prueba

Fundamentos de Automatización de la prueba

Introducción a 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 la 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:

* 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.

Desventajas:

* 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.
* Nivel de tecnisismo.

Patrones de diseño

Patrones de diseño: 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:*](#PageObjectModel)

* 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 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.

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.

*Objetivos a tener en cuenta:*

* 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.

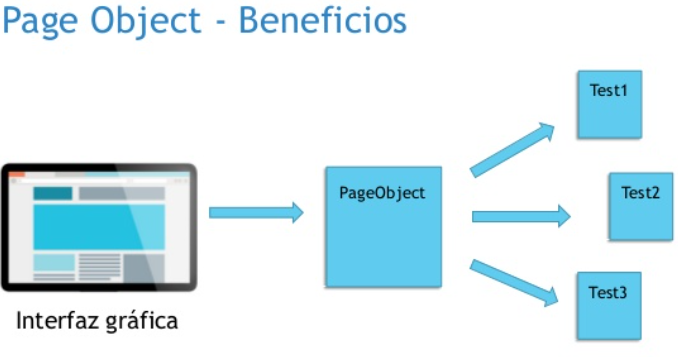
En definitiva, desarrollar un software de calidad.

Page Object Model: También llamado POM, es la implementación del patrón de diseño 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 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. Estas interacciones generalmente se dividen en acciones y validaciones. Las *acciones* reflejarán aquellos pasos que el usuario debe realizar dentro del sitio web o aplicación que se encuentra bajo prueba. En cambio, las *validaciones* serán métodos que reflejen la confirmación de que el sistema se comporta como se espera luego de una acción específica.

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



Introducción a Selenium: 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 IDE:*](#SeleniumIDE)Es una herramienta de automatización que nos permite grabar, editar y depurar pruebas, sin la necesidad del uso de un lenguaje de programación. También se lo conoce como Selenium Recorder.

*Selenium WebDriver:* Herramienta que permite automatizar pruebas UI (User Interface) o interfaz de usuario de aplicaciones web. Algunos de los lenguajes que soportan Selenium WebDriver Java, C#, Python, Ruby, PHP, JavaScript.

Es útil para poder simular la manera en que los usuarios reales interactúan con alguna aplicación web. Para ello, provee una serie de métodos para accionar y validar cualquier elemento dentro de una interfaz gráfica.

*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. Cuenta con dos componentes: Selenium Hub y Remote Control.

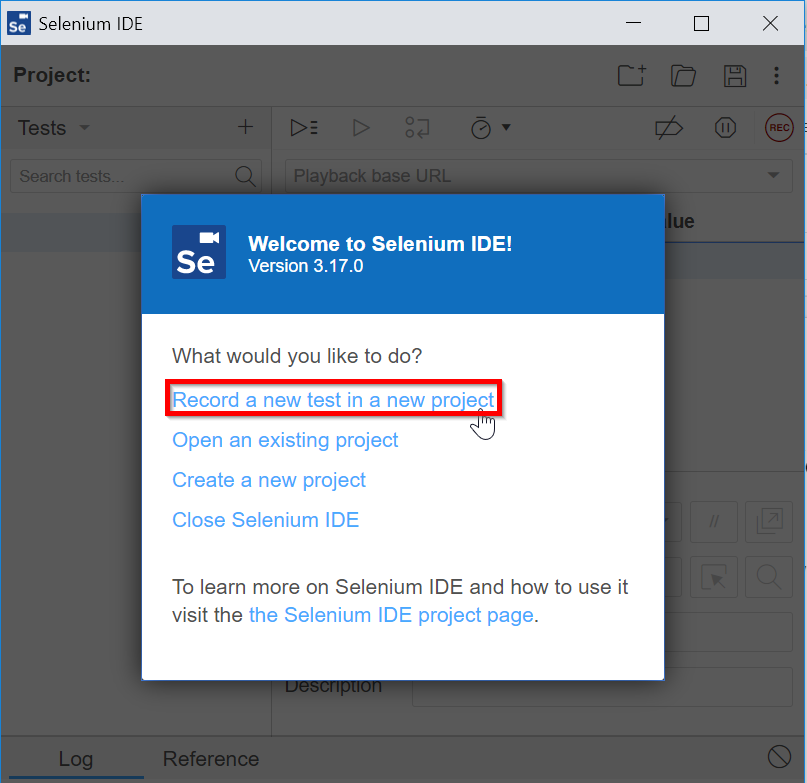
Selenium IDE: Se trata de un entorno de pruebas de software para aplicaciones basadas en la web que permite realizar tareas de Record&Play de flujos de pruebas. Los flujos grabados quedan contenidos en un script que se puede editar y parametrizar para adaptarse a los diferentes casos, su ejecución se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

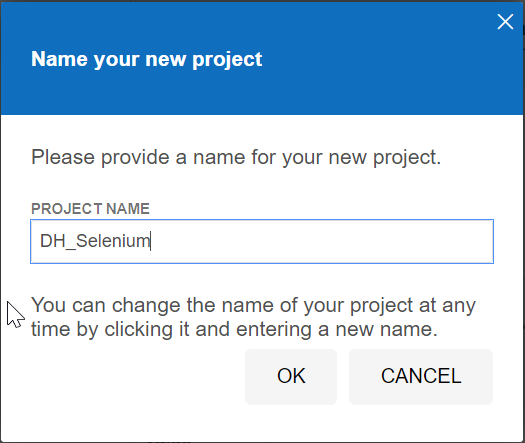
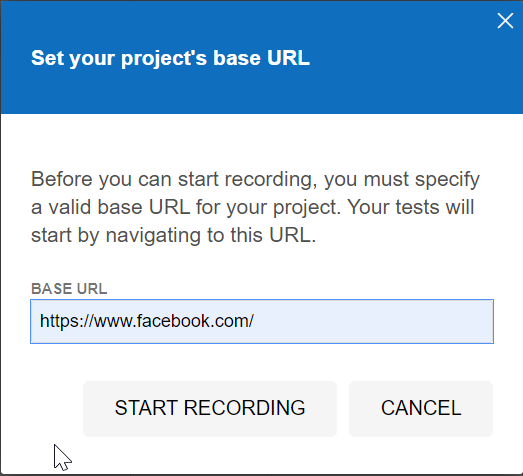
Su principal objetivo es automatizar pruebas funcionales repetitivas para luego facilitar el trabajo del tester, como también pruebas de regresión.​

Permite referenciar objetos del DOM en base al ID, CSS, nombre o a través de XPath.​

Las acciones pueden ser ejecutadas paso a paso.

Ejemplo Record & Play:

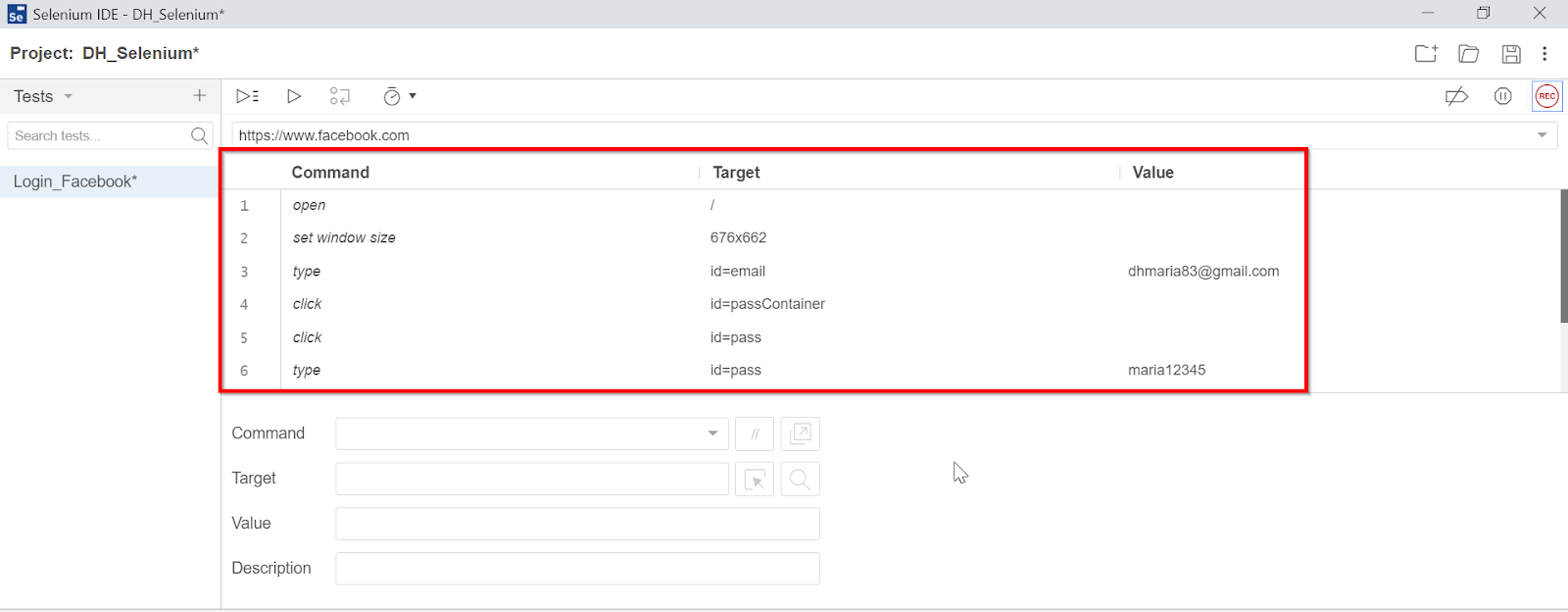
Vamos a crear/grabar el caso de prueba de login de Facebook. Para ello debemos realizar los siguientes pasos:

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 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.

Vidio tutorial: <https://drive.google.com/file/d/1ySkGFuQHKxoS8nExuQU7wkmj0YQvgVk7/view?usp=sharing>

En la imagen se visualizan:

* *Comandos:* son 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 y obtenido durante la grabación. También se puede establecer un nuevo valor para pruebas siguientes.

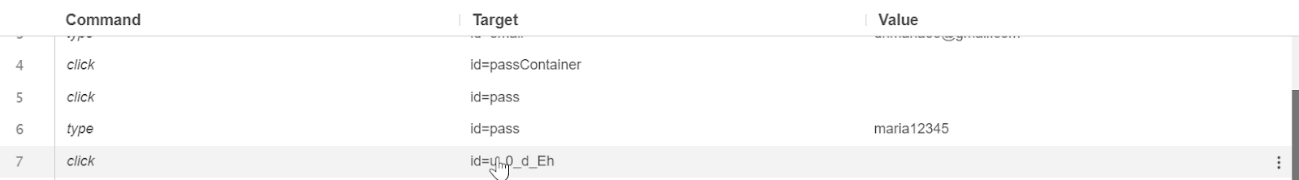


Hay distintas formas de localizar los elementos en la web. Puede ser por id o name entre otros. En el ejemplo del login vemos que se localizan por el id:



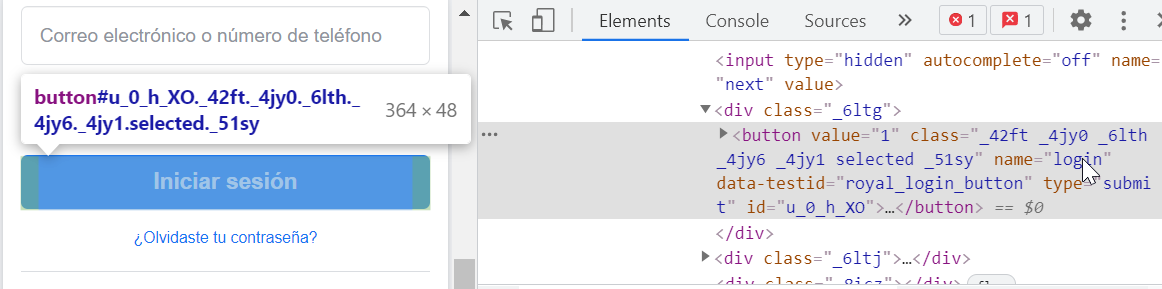
Podemos observar que se localizó el botón de inicio de sesión con id. Al reproducir este caso de prueba, una vez que llega a ejecutarlo entra en un lapso sin dar respuesta. Por lo tanto, se asume que se localizó de manera incorrecta el elemento, en este caso, el botón de inicio de sesión.

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

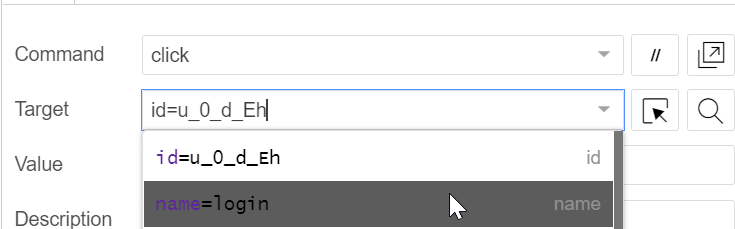


Se debe inspeccionar el elemento y validar, si es posible, la localización por otro parámetro. En este caso, se puede localizar por name. Entonces, en Selenium se debe modificar el target a name:

Web:



Selenuim:



## C23A – Automatización de la prueba

Selenium WebDriver

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. Además también permite realizar validaciones en el comportamiento de cualquier componente. Esto nos sirve para verificar el comportamiento esperado que se encuentra detallado en los casos de prueba diseñados y listos para ser automatizados.

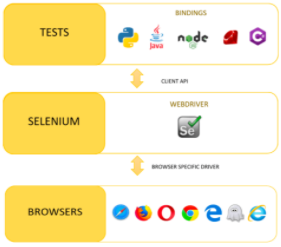
WebDriver utiliza las API de automatización del navegador proporcionadas por los desarrolladores de los navegadores para controlarlo y ejecutar pruebas. Esto es como si un usuario real estuviera manipulándolo con la principal diferencia en la velocidad de ejecución, WebDriver es considerablemente más veloz que una ejecución manual.

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:



1. El happy path, es un tipo de prueba positiva y que busca validar el flujo normal de un sistema. Por ejemplo, si tenemos que testear un Login, ¿cuál sería el camino ideal? Que el usuario ingrese un nombre de usuario y contraseña correctos, haga clic en el botón loguearse y que la aplicación se abra. Este camino ideal, es lo que conocemos como happy path. Son los primeros test que deberíamos hacerle a nuestra aplicación.

2. Si, al testing positivo y negativo lo podemos categorizar como una modalidad de casos de prueba.

3. En el día a día, los tester generalmente hacemos pruebas en el ambiente de QA o de producción. Por ahora, esta correcto que utilicen el ambiente de QA para el diseño de los casos de prueba. No le den muchas vueltas, vamos a profundizar el tema de los ambientes más adelante en el cursado

4. Con respecto a la severidad y prioridad, es más que nada una evaluación subjetiva y lo vas mejorando con los años. Pero por ejemplo si no me puedo loguear en la aplicación y esto lo tienen que solucionar si o si, sino el usuario está bloqueado. Entonces su severidad y prioridad es alta. En cambio, si por ejemplo al ingresar mal la contraseña me sale un mensaje de error y no lo puedo cerrar pero si me puedo loguear, esto tiene una severidad baja, porque puedo seguir trabajando con la app.