

## CLASE 1 SEMANA 1

Testing = Calidad.

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. 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:

- 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.

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

**Aspecto psicológico del testing** → nuestro objetivo debe ser demostrar que un programa tiene errores, así nuestros datos de prueba tendrán una mayor probabilidad de encontrarlos.

#### Independencia de la prueba →

Ventajas:

+Es probable que los probadores independientes reconozcan diferentes tipos de fallos en comparación con los desarrolladores debido a sus diferentes contextos, perspectivas técnicas y sesgos.

+Un probador independiente puede verificar, cuestionar o refutar las suposiciones hechas por los implicados durante la especificación e implementación del sistema.

Desventajas:

-Los desarrolladores pueden perder el sentido de la responsabilidad con respecto a la calidad

-Los probadores independientes pueden ser vistos como un cuello de botella o ser culpados por los retrasos en el lanzamiento o liberación.

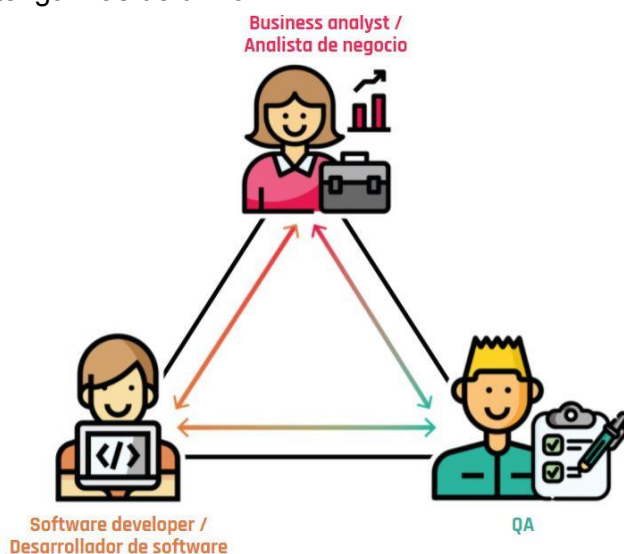
-Los probadores independientes pueden carecer de información importante --por ejemplo, sobre el objeto de prueba--.

#### 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.



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

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

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

### **Ciclo de vida de las pruebas de software**

Ciclo de Deming → Ciclo PHVA → Metodología más usada para solucionar problemas y ejecutar sistemas de mejora continua. Su aplicación ayuda a que las organizaciones mejoren su rendimiento y aumenten su productividad.

Cuatro etapas cíclicas de forma que una vez acabada la etapa final se debe volver a la primera y repetir el ciclo de nuevo. Planificar, Hacer, Verificar y Actuar.

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

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

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

Ambos persiguen el mismo objetivo → La entrega de un producto de calidad, a través de la mejora continua de sus procesos.

-Se pueden ubicar las actividades del STLC en el ciclo de Deming de la siguiente forma:



**-Planificación** → Se definen los objetivos y el enfoque de la prueba dentro de las restricciones impuestas por el contexto.

Algunas subactividades realizadas son:

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

Documentos de salida:

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

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

Algunas subactividades realizadas son:

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

Documento de salida:

- Informe de avance de la prueba.

**-Análisis** → Durante esta actividad se determina “qué probar”.

Algunas subactividades realizadas son:

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

Documento de salida:

- \*Contratos de prueba que contienen las condiciones de prueba.

**-Diseño** → Durante esta actividad se determina “cómo probar”

Algunas subactividades realizadas son:

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

Documento de salida:

- \*Casos de prueba de alto nivel diseñados y priorizados.

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

Algunas subactividades realizadas son:

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

Documentos de salida:

- \*Procedimientos y datos de prueba
- \*Calendario de ejecución
- \*Test suite.

**-Ejecución** → Durante esta actividad se realiza la ejecución de los casos de prueba.

Algunas subactividades realizadas son:

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

Documentos de salida:

- \*Reporte de defectos
- \*Informe de ejecución de pruebas.

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

Algunas subactividades realizadas son:

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

Documentos de salida:

- \*Informe resumen de prueba
- \*Lecciones aprendidas.

## CLASE 2 SEMANA 1

Desde el análisis a la implementación  
Enfoque tradicional vs Enfoque ágil.



### Niveles de prueba

Para desarrollar una aplicación de software hay que llevar a cabo pruebas en todos los niveles para entregar al cliente un producto que cumpla con todas sus expectativas y necesidades asegurando la calidad del mismo. Para entregar un producto de calidad y confiable hay que testearlo.

Existen distintos niveles de prueba o actividades que se organizan y gestionan conjuntamente.

Modelo en V → Modelo empleado en diversos procesos de desarrollo.

Para cada fase del desarrollo (izquierda) debe existir un resultado verificable, medidas de control de calidad (derecha). En la unión de ambos lados, se sitúa la implementación del producto. Diferentes niveles de control de la calidad.

-Verificación → conjunto de actividades que aseguran que el software implementa correctamente una función específica.

¿Estamos construyendo el producto correctamente?

-Validación → es un conjunto diferente de actividades que aseguran que un software construido respeta los requisitos del cliente.

¿Estamos construyendo el producto correcto?

Abarca las pruebas de componente, de integración, de sistema y de aceptación.

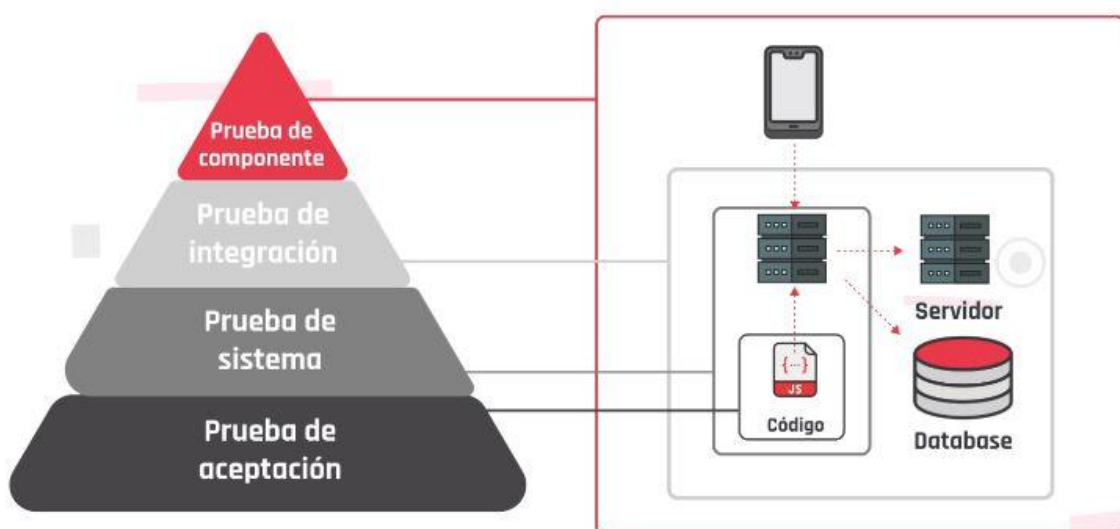


- **De Componente** → se aísla cada parte del programa clase, módulo, objetos, paquetes, sistemas para comprobar que cada una de esas partes funciona correctamente por separado.

- **De integración** → Se realizan sobre un conjunto de módulos de la aplicación para comprobar que funciona correctamente. Se prueba la interacción entre las distintas partes del software.

- **De sistemas** → son llevadas a cabo por el proveedor en un entorno de laboratorio para demostrar que el sistema cumple con los requisitos funcionales y no funcionales, y con el diseño técnico. Están diseñadas para probar el sistema en su totalidad.

- **De aceptación** → realizadas por el usuario en un entorno muy similar al de producción para demostrar que el sistema cumple las especificaciones funcionales y requisitos del cliente.





Los niveles de prueba se caracterizan por los siguientes atributos:

- Objetivos específicos.
- Bases de prueba. → ejemplos de productos de trabajo que se pueden utilizar como base de prueba.
- Objeto de prueba.
- Defectos y fallos característicos.
- Enfoques y responsabilidades específicas.

## **NIVELES DE PRUEBA**

### **-Pruebas de componente:**

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

\*Base de prueba → Diseño detallado. Código. Modelo de datos. Especificaciones de los componentes.

\*Objetos de prueba → Componentes. Unidades o módulos. Código y estructuras de datos. Clases. Módulos de base de datos.

\*Defectos y fallos característicos → Funcionamiento incorrecto. Problemas de flujo de datos. Código y lógica incorrectos.

\*Enfoque y responsabilidades específicas → El desarrollador que escribió el código realiza esta prueba. Estos escriben y ejecutan pruebas después de haber escrito el código de un componente. Pero en el desarrollo ágil, la redacción de casos de prueba de componente automatizados puede preceder a la redacción del código de la aplicación.

**-Pruebas de integración:** Se centra en las interacciones entre componentes o sistemas.

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

\*Base de prueba → Diseño de software y sistemas. Diagramas de secuencia. Especificaciones de interfaz y protocolos de comunicación. Casos de uso. Arquitectura a nivel de componente o de sistema. Flujos de trabajo. Definiciones de interfaces externas.

\*Objetos de prueba → Subsistemas. Bases de datos. Infraestructura. Interfaces. Interfaces de programación de aplicaciones (API). Microservicios.

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

\*Enfoque y responsabilidades específicas → La prueba de integración debe concentrarse en la integración propiamente dicha. Se puede utilizar los tipos de prueba funcional, no funcional y estructural. Responsabilidad de los testers.

### **-Prueba de sistema:**



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

\*Base de prueba → Especificaciones de requisitos del sistema y del software (funcionales y no funcionales). Informes de análisis de riesgo. Casos de uso. Épicas e historias de usuario. Modelos de comportamiento del sistema. Diagramas de estado. Manuales del sistema y del usuario.

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

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

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

**-Prueba de aceptación:**, al igual que la prueba del sistema, se centra normalmente en el comportamiento y las capacidades de todo un sistema o proyecto.

\*Objetivos → Establecer confianza en la calidad del sistema en su conjunto. Validar que el sistema está completo y que funcionará como se espera. Verificar que los comportamientos funcionales y no funcionales del sistema sean los especificados.

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

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

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

\*Enfoque y responsabilidades específicas → Responsabilidad de los clientes, usuario de negocio, propietarios de producto u operadores de un sistema, y otros implicados también pueden estar involucrados. La prueba de aceptación se considera, a menudo, como el último nivel de prueba en un ciclo de vida de desarrollo secuencial.

### TIPOS DE PRUEBA:

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

Objetivos:

\*Evaluar las características de calidad funcional tales como la completitud, corrección y pertinencia.

\*Evaluar características no funcionales de calidad, tales como la fiabilidad, eficiencia de desempeño, seguridad, confiabilidad y usabilidad.

Prueba Funcional	Prueba no Funcional
La prueba funcional de un sistema incluye pruebas que evalúan las funciones que el sistema debe realizar. Las funciones describen qué hace el sistema.	La prueba no funcional prueba “cómo de bien” se comporta el sistema.
La prueba funcional observa el comportamiento del software.	El diseño y ejecución de la prueba funcional puede implicar competencias y conocimientos especiales, como el conocimiento de las debilidades inherentes a un diseño o tecnología —por ejemplo: vulnerabilidades de seguridad asociadas con determinados lenguajes de programación—.

### CLASE 4 SEMANA 2

#### DEFECTO:

Los 3 conceptos están relacionados porque uno depende del otro.

**Error:** se produce por la equivocación de una persona. Error del programador.

**Defecto:** Esto causa un defecto en el software.

**Fallo:** produce un fallo cuando la prueba se ejecuta.

El desarrollador al escribir el código lo hace incorrectamente cometiendo un error, eso se convierte en algo incorrecto en el código, un defecto, y al ejecutarse produjo un mal funcionamiento, un fallo.

Nuestro objetivo como probadores es brindar calidad dentro de un sistema de software.

**Ciclo de vida de un defecto:** El proceso que gestiona un defecto desde su descubrimiento hasta su solución. En cada estado sólo existe un responsable del defecto, excepto en estados terminales (cerrado, duplicado), debido a que no se van a realizar más acciones.

-Nuevo/Inicial → Se recopila información y se registra el defecto.

-Asignado → Si es un defecto válido y debe solucionarse se asigna al equipo de desarrollo, sino se puede rechazar o diferir. Duplicado → Si el defecto se repite o existe otro con una misma causa raíz. Devuelto o rechazado → Se solicita más información o el receptor

rechaza el defecto. Diferido → el defecto no es prioritario y se solucionará en una próxima versión.

-En progreso → Se analiza y trabaja en la solución.

-Corregido → Se realizan los cambios de código para solucionar el defecto.

-En espera de verificación → En espera de que sea asignado a un probador. El desarrollador está a la expectativa del resultado de la verificación.

-En verificación → El probador ejecuta una prueba de confirmación. Reabierto → debe contener la siguiente descripción “La prueba de confirmación indica que el defecto no se ha solucionado”

-Verificado → Se obtiene el resultado esperado en la prueba de confirmación.

-Cerrado → El defecto fue corregido y se encuentra disponible para el usuario final.

### **GESTIÓN DE DEFECTOS:**

\*Proceso general → 1) Detectar 2) Registrar → Varía según el contexto del componente o sistema, nivel de prueba y modelo de desarrollo elegido. 3) Investigación y seguimiento. 4) Clasificación/Resolución.

Objetivos:

-Brindar información sobre cualquier evento adverso que haya ocurrido, para poder identificar efectos específicos, aislar el problema con una prueba de reproducción mínima y corregir los defectos potenciales.

-Proporcionar a los jefes de prueba un medio para un seguimiento de la calidad del producto de trabajo y del impacto en la prueba.

-Dar ideas para la mejora de los procesos de desarrollo y prueba.

\*Escribir un informe de defectos → Si el defecto se reporta eficientemente, las probabilidades de que sea solucionado rápidamente es mayor. La solución de un defecto dependerá de la eficiencia con que se reporte.

¿Qué condiciones debemos tener en cuenta?

-Los bugs deben tener identificadores únicos.

-Una falla debe ser reproducible para reportarla.

-Ser específico.

-Reportar cada paso realizado para reproducirlo.

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

-Redactar un defecto de manera excesivamente coloquial y ambigua.

-Dar solo una captura del defecto sin indicar qué se estaba haciendo cuando sucedió.

-No incluir en la descripción del defecto cuál era el resultado esperado para los pasos realizados.

-No determinar un patrón con el cual el defecto ocurre antes de reportar el mismo.

-No leer el defecto reportado siguiendo los pasos uno mismo para ver que la descripción es clara.

-No incluir información que dada las características del defecto, la misma es de relevancia.

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 3 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.

### Partes de un informe de defectos:

Atributo	Descripción	Ejemplo
<b>ID</b>	Abreviatura de identificador, un código único e irrepetible que puede ser número o letras.	001 - Test01
<b>Título</b>	El título debe ser corto y específico, que se entienda en este lo que queremos reportar. Cuando el desarrollador o el equipo vean el título pueden interpretar rápidamente qué es, dónde está y cuán importante es ese defecto.	Login - Ingresar con campos en blanco
<b>Descripción</b>	Describir un poco más sobre el error, es decir, desarrollar lo que dejamos afuera en el título lo podríamos explicar acá.	En la pantalla login si dejo vacío los campos nombre y password y apretó ingresar, me lleva a la página principal.
<b>Fecha del informe del defecto</b>	La fecha que detectó el defecto para saber posteriormente el tiempo en que se resolvió.	23/04/21
<b>Autor</b>	El nombre del tester que descubrió el defecto, por si el desarrollador tiene una duda, sabe a quién consultar.	Pepito Román
<b>Identificación del elemento de prueba</b>	Nombre de la aplicación o componente que estamos probando.	Carrito compras
<b>Versión</b>	Es un número que nos indica en qué versión está la aplicación.	1.0.0

Atributo	Descripción	Ejemplo
<b>Entorno</b>	El entorno en el que probamos (desarrollo, QA, producción).	Desarrollo
<b>Pasos a reproducir</b>	Los pasos a seguir para llegar al defecto encontrado.	1) Ingresar a la aplicación. 2) Dejar en blanco el campo nombre. 3) Dejar en blanco el campo password. 4) Hacer click en el botón "Ingresar".
<b>Resultado esperado</b>	Es lo que esperamos que suceda o muestre la aplicación muchas veces según los requerimientos de la misma.	No debe ingresar a la aplicación sin un usuario y una contraseña válidos.
<b>Resultado obtenido o actual</b>	Es lo que sucedió realmente o lo que nos mostró la aplicación. Puede coincidir o no con el resultado esperado, si no coincide, hemos detectado un error o bug.	Ingresa a la aplicación sin usuario y sin contraseña.
<b>Severidad</b>	Cuán grave es el defecto que hemos encontrado, puede ser: bloqueado, crítico, alto, medio, bajo o trivial.	Crítico
<b>Prioridad</b>	Con esto decimos qué tan rápido se debe solucionar el defecto, puede ser: alta, media, baja	Alta
<b>Estado del defecto</b>	Los estados pueden ser: nuevo, diferido, duplicado, rechazado, asignado, en progreso, corregido, en espera de verificación, en verificación, verificado, reabierto y cerrado.	Nuevo

Atributo	Descripción	Ejemplo
<b>Referencias</b>	Link al caso de prueba con el cual encontramos el error.	<a href="https://repositorio.-com.ar/TC-001-User-Login">https://repositorio.-com.ar/TC-001-User-Login</a>
<b>Imagen</b>	Se puede adjuntar una captura de pantalla del error, esto nos permite demostrar que el error sucedió y al desarrollador lo ayuda a ubicar el error.	

## CLASE 5 SEMANA 2

### DISEÑO DE LA PRUEBA

**CASOS DE PRUEBA** → Es un documento escrito que proporciona información escrita sobre qué y cómo probar.

#### Características de un buen caso de prueba:

- No asumir → no asumir la funcionalidad y las características de la aplicación mientras se prepara el caso de prueba. Se debe ser fiel a los documentos de especificación y ante cualquier duda, hay que consultar.
- Asegurar la mayor cobertura posible → Escribir casos de prueba para todos los requisitos especificados.
- Autonomía → El caso de prueba debe generar los mismos resultados siempre, sin importar quien lo pruebe.
- Evitar la repetición de casos de prueba → Si se necesita un caso de prueba para ejecutar otro, indicar caso de prueba por su ID.
- Deben ser simples → Se deben crear casos de pruebas que sean lo más simples posibles ya que otra persona que no sea el autor puede ejecutarlos. Utilizar un lenguaje asertivo para facilitar la comprensión y que la ejecución sea más rápida.
- El título debe ser fuerte → Solo leyendo el título, cualquier probador debería comprender el objetivo del caso de prueba.
- Tener en cuenta al usuario final → El objetivo final es crear casos de prueba que cumplan con los requisitos del cliente y que sean fáciles de usar.

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

- Identificador → Puede ser numérico o alfanumérico. La mayoría de herramientas lo generan automáticamente.
- Nombre del caso de prueba (conciso) → Se debe utilizar una nomenclatura que esté definida, pero, si no existe, lo recomendable es incluir el nombre de módulo al que corresponde el caso de prueba.
- Descripción → Debe decir qué se va a probar, el ambiente de pruebas y los datos necesarios para ejecutarlo.

#### -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 - Happy path:

- +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 - Happy path:

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

### **Caso de uso - Caso de prueba**

**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:



$$\text{Cobertura} = \frac{\text{Comportamientos o rutas del caso de uso probadas}}{\text{Comportamientos o rutas del caso de uso totales}}$$

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

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

### CLASE 7 SEMANA 3

#### **TÉCNICAS DE PRUEBA**

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

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

-**Técnicas basadas en la experiencia** → Aprovechan el conocimiento de desarrolladores, probadores y usuarios para diseñar, implementar y ejecutar las pruebas.

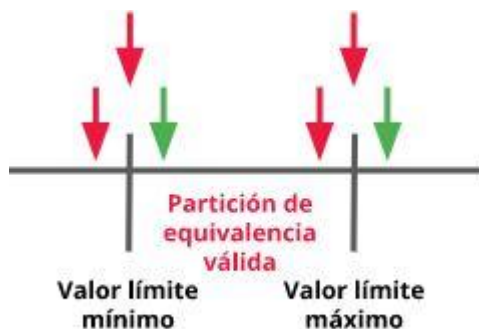
## TÉCNICAS DE CAJA NEGRA →

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

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

$$\text{Cobertura} = \frac{\text{Particiones probadas}}{\text{Particiones identificadas}}$$

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



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

$$\text{Cobertura} = \frac{\text{Valores límites probados}}{\text{Valores límites identificados}}$$

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

Condiciones	Reglas							
Condición 1	S	S	S	S	N	N	N	N
Condición 2	S	S	N	N	S	S	N	N
Condición 3	S	N	S	N	S	N	S	N
Acción 1	X	X						
Acción 2				X		X		X
Acción 3			X				X	
Acción 4					X			

\*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:

$$\text{Cobertura} = \frac{\text{Número de reglas de decisión probadas}}{\text{Número de reglas de decisión totales}}$$

### 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 como no funcionales.

## CLASE 8 SEMANA 3

### EJECUCIÓN DE LOS CASOS DE PRUEBA

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

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

### CREACIÓN DE SUITES →

Durante el 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.

## CLASE 10 SEMANA 4

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

**-Pruebas estáticas** → Se basa en la evaluación manual de los productos de trabajo (revisiones) o en la evaluación basada en herramientas del código u otros productos de trabajo (análisis estático).

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

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

### Ventajas de las pruebas estáticas tempranas:

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

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

### Defectos encontrados con pruebas estáticas

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

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

**-Pruebas dinámicas** → Las pruebas dinámicas requieren la ejecución del software, componente o sistema.

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

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

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

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

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

### Proceso de revisión:

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

Las revisiones pueden ser:

**\*Revisiones formales:** Tienen roles definidos, siguen un proceso establecido y deben ser documentadas.

**\*Revisiones informales:** No siguen un proceso definido y no son documentadas formalmente.

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



#### -Roles >

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

#### -Tipo >

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

#### -Técnicas >

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

#### -Actividades >



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

## **Requerimientos**

Título: Requisitos

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

**-Requisitos funcionales:** Definen lo que un sistema permite hacer desde el punto de vista del usuario. Estos requisitos deben estar especificados de manera explícita. Ejemplo: El campo de monto acepta únicamente valores numéricos con dos decimales (pruebas funcionales y de sistema).

**-Requisitos no funcionales:** Definen condiciones de funcionamiento del sistema en el ambiente operacional. Ejemplos:

\*Requisito de usabilidad → La usabilidad se define como el esfuerzo que necesita hacer un usuario para aprender, usar, ingresar datos e interpretar los resultados obtenidos de un software de aplicación (pruebas de usabilidad).

\*Requisito de eficiencia → Relacionado con el desempeño en cuanto al tiempo de respuesta, número de operaciones por segundo, entre otras mediciones, así como consumo de recursos de memoria, procesador y espacio en disco o red (pruebas de rendimiento, pruebas de carga, estrés y escalabilidad, pruebas de gestión de la memoria, compatibilidad e interoperabilidad).

\*Requisito de disponibilidad → Disposición del sistema para prestar un servicio correctamente (pruebas de disponibilidad).

\*Requisito de confiabilidad → Continuidad del servicio prestado por el sistema (pruebas de seguridad).

\*Requisito de integridad → Ausencia de alteraciones inadecuadas al sistema (pruebas de seguridad, pruebas de integridad).

\*Requisito de mantenibilidad → Posibilidad de realizar modificaciones o reparaciones a un proceso sin afectar la continuidad del servicio (pruebas de mantenimiento y de regresión).

## **CLASE 11 SEMANA 4**

### **ENTORNOS DE PRUEBA:**

## Tipos de prueba según ambiente



### 1-DEV Development

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

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

### 2-QA Test

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

**-Pruebas de casos de uso** → Proporcionan pruebas transaccionales, basadas en escenarios, que deberían emular el uso del sistema.

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

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

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

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

**-Pruebas de confirmación** → Consiste en volver a ejecutar los pasos para reproducir el fallo o los fallos causados por un defecto en la nueva versión de software, una vez corregido

el defecto, para así confirmar que el defecto original se ha solucionado satisfactoriamente o detectar efectos secundarios no deseados.

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

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

### 3-UAT User Acceptance Testing

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

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

**-Pruebas de usabilidad** → Evalúan la facilidad con la que los usuarios pueden utilizar o aprender a utilizar el sistema para lograr un objetivo específico en un contexto dado.

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

### 4-STG Staging

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

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

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

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

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

**-Pruebas de gestión de la memoria** → Evalúan el estado y la integridad de la memoria del sistema para identificar problemas potenciales.

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

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

**-Pruebas de migración de datos** → Incluyen las pruebas realizadas al transferir datos entre tipos de dispositivos de almacenamiento, formatos o sistemas de cómputos.

## **MÉTRICA Y REPORTE**

Lo que no se puede medir no se puede controlar; lo que no se puede controlar no se puede gestionar; lo que no se puede gestionar no se puede mejorar.

Peter Ducker.

Las métricas de pruebas pueden clasificarse en las siguientes categorías:

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

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

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

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

El uso de métricas permite a los probadores comunicar resultados de una manera consistente y llevar a cabo un seguimiento coherente del progreso a lo largo del tiempo. Existen 5 dimensiones primarias sobre las que se monitoriza el progreso de las pruebas:

-Riesgos (de la calidad) del producto

-Defectos

-Pruebas

-Cobertura

-Confianza

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

-El estado de las actividades de prueba y el avance con respecto al plan de prueba.

-Factores que impiden el avance.

-Pruebas previstas para el próximo período objeto del informe.

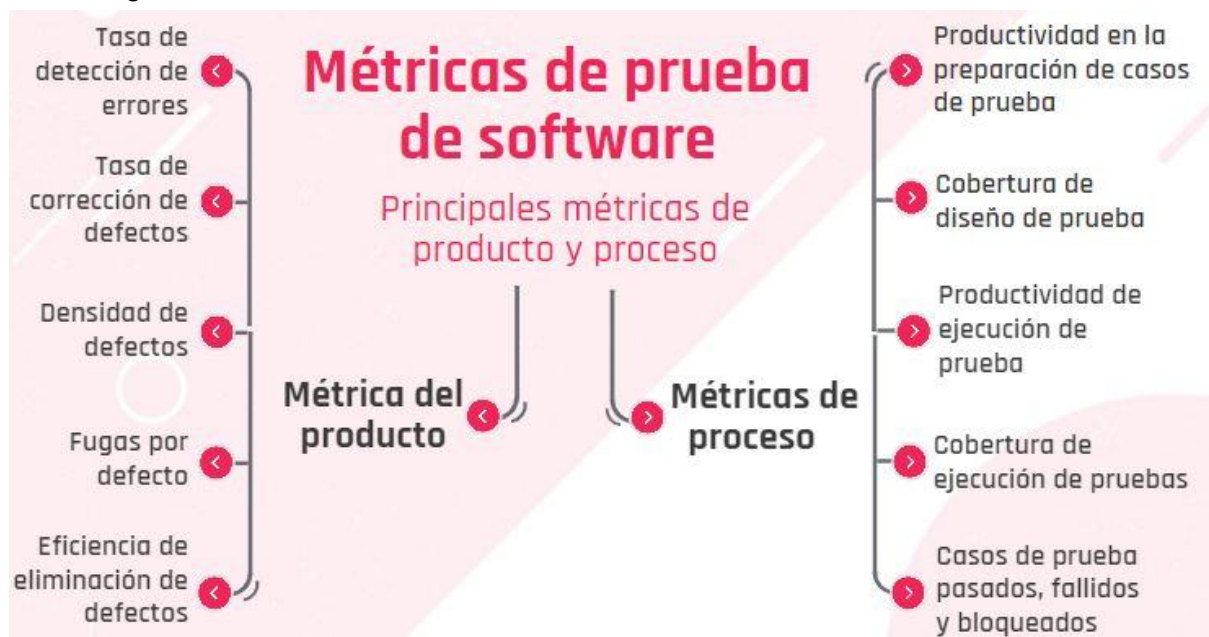
-La calidad del objeto de prueba.

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

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

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

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



**-Métricas del producto** → Métricas utilizadas en el proceso de análisis de defectos del ciclo de vida de las pruebas de software.

### Tasa de detección de errores

Es para determinar la efectividad de los casos de prueba.

$$\text{Tasa de detección de errores} = \left( \frac{\text{Nro. total de defectos encontrados}}{\text{Nro. total de casos de prueba ejecutados}} \right) \times 100$$

## Tasa de corrección de defectos

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

$$\text{Tasa de corrección de defectos} = \left\{ \frac{\text{Nro total de defectos notificados como solucionados} - \text{Nro total de defectos reabiertos}}{\text{Nro total de defectos notificados como solucionados} + \text{Nro. total de nuevos errores debido a la corrección}} \right\} \times 100$$

## Densidad de defectos

Se define como la relación entre defectos y requisitos.

$$\text{Densidad de defectos} = \frac{\text{Nro. total de defectos identificados}}{\text{Tamaño real (cantidad de requerimientos)}}$$

## Fugas por defecto

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

$$\text{Fuga por defecto} = \left( \frac{\text{Nro. de defectos encontrados en UAT}}{\text{Nro. de defectos encontrados antes de UAT}} \right) \times 100$$

## Eficiencia de eliminación de defectos

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

$$\text{Eficiencia de eliminación de defectos} = \left\{ \frac{\text{Nro. total de defectos encontrados antes de la entrega}}{\text{Nro. total de defectos encontrados antes de la entrega} + \text{Nro. total de defectos encontrados después de la entrega}} \right\} \times 100$$

**-Métricas de proceso** → Métricas utilizadas en el proceso de preparación y ejecución de pruebas del ciclo de vida de las pruebas de software.



## Productividad en la preparación de casos de prueba

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

$$\text{Productividad de preparación de casos de prueba} = \frac{\text{Nro. de casos de prueba}}{\text{Esfuerzo invertido en la preparación del caso de prueba (en horas)}}$$

## Cobertura de diseño de prueba

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

$$\text{Cobertura de diseño de prueba} = \frac{\text{Número total de requisitos asignados a casos de prueba}}{\text{Número total de requisitos}}$$

## Productividad de ejecución de prueba

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

$$\text{Productividad de ejecución de prueba} = \frac{\text{Nro. de casos de prueba ejecutados}}{\text{Esfuerzo gastado para la ejecución de casos de prueba}}$$

## Cobertura de ejecución de pruebas

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

$$\text{Cobertura de ejecución de prueba} = \frac{\text{Nro. total de casos de prueba planeados para ejecutar}}{\text{Nro. total de casos de prueba ejecutados}}$$

## Casos de prueba pasados, fallidos y bloqueados

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

$$\text{Casos de prueba pasados} = \frac{\text{Nro. total de casos de prueba pasados}}{\text{Nro. total de casos de prueba ejecutados}}$$

$$\text{Casos de prueba fallidos} = \frac{\text{Nro. total de casos de prueba fallidos}}{\text{Nro. total de casos de prueba ejecutados}}$$

$$\text{Casos de prueba bloqueados} = \frac{\text{Nro. total de casos de prueba bloqueados}}{\text{Nro. total de casos de prueba ejecutados}}$$



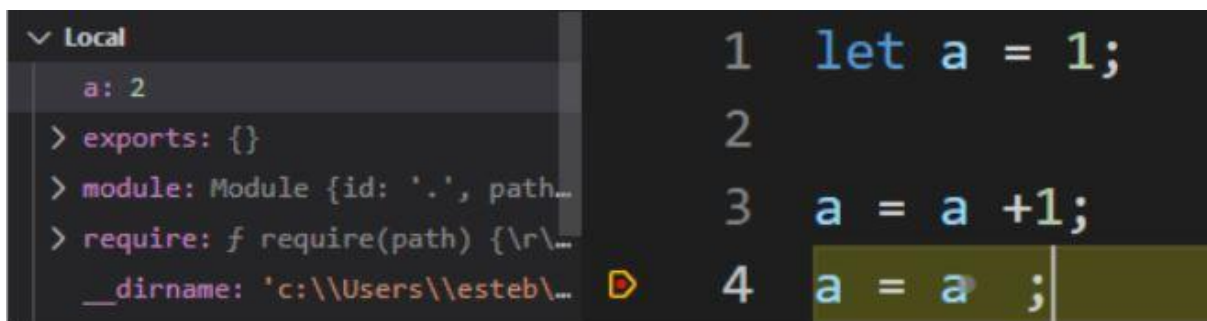
## CLASE 14 SEMANA 5

### INTRODUCCIÓN A LA PRUEBA DE COMPONENTE:

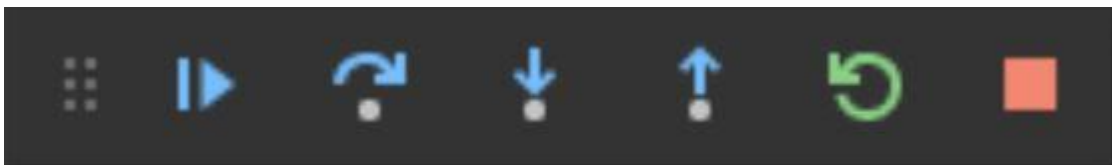
**-Debug o depurar** → proceso de encontrar, analizar y remover las causas de fallos en el software. Lo que se realiza es la ejecución paso a paso de cada instrucción del programa para analizar las variables y sus valores.

**-Breakpoint** → pausa en nuestro código para entender en qué estado están nuestras variables.







-Control de variables: En el breakpoint nuestra variable vale “2” → se refleja en el menú de debug de la izquierda.



**Interfaz:** Nos despliega el siguiente menú:



Controles:

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

**Debug desde la consola de Chrome:**

Es posible depurar código JavaScript directamente desde la consola de Chrome:

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

Continúa abajo

## Debugging



VS.

## Testing



Proceso deductivo para corregir los errores encontrados durante las pruebas.

Permite dar solución a la falla del código.

Es la investigación y detección del error.

Realizado generalmente por el programador o el desarrollador, a excepción del código generado por el probador como parte de los script de prueba automáticos.

No se puede realizar sin el conocimiento de diseño adecuado.

Lo realiza únicamente un interno. Generalmente, un externo no puede depurar debido a que no debería tener acceso al código.

Realizado en forma manual.

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.

No es un aspecto del ciclo de vida del desarrollo de software, ocurre como consecuencia de las pruebas.

Busca hacer coincidir el síntoma con la causa, lo que conduce a la corrección del error.

Comienza con la ejecución del código debido a un caso de prueba fallido.

Proceso donde se comprueba que el sistema o componente funcione de acuerdo a lo esperado, tiene como objetivo la búsqueda de errores.

Permite identificar la/s falla/s del código implementado.

Es la visualización de errores.

Realizado generalmente por el probador.

No es necesario tener conocimientos de diseño en el proceso de prueba.

Puede ser realizado tanto por personas internas como externas.

Puede ser manual o automatizado.

Basado en diferentes niveles de prueba, es decir, pruebas de componentes, pruebas de integración, pruebas del sistema, etc.

Etapas del ciclo de vida del desarrollo de software (SDLC).

Compuesto por la validación y verificación del software.

Iniciado antes de tener el código (testing estático) o después de que se escribe el código (testing dinámico).

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

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

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

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

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

### ¿Que es una “unidad”?

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

## Framework para pruebas unitarias

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

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

- Mocha (<https://mochajs.org/api/index.html>)

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

## Desarrollo guiado por pruebas (TDD)

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

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

## CLASE 16 SEMANA 6

### PRUEBA DE COMPONENTE:

Unit Testing / Prueba de componentes:

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

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

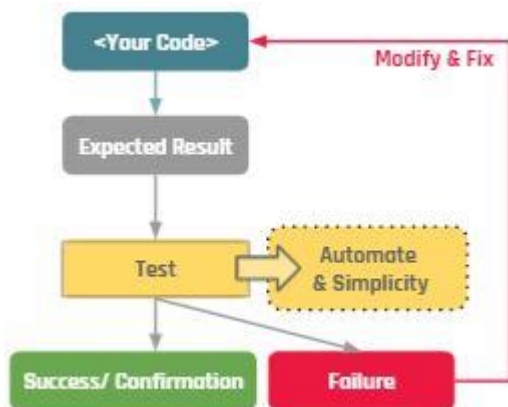
Este tipo de pruebas puede cubrir:

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.

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.
  - a) Si el test pasa, se confirma el resultado esperado.
  - b) 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:** Estas son algunas de las ventajas de realizar pruebas de componente o unit test dentro de un proyecto:

- 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 2 tipos:

- Manuales → se puede emplear un documento instructivo paso a paso.
- Automatizadas → Se necesita de un framework automatizado para escribir los scripts de prueba.

¿Qué se necesita para automatizar los unit test?

- Test Runner → Es una herramienta que ejecuta los test y muestra los resultados en forma de reporte. Ejemplo: Mocha
- Assertion Liberty → Es una herramienta que se utiliza para validar la lógica de prueba, las condiciones y resultados esperados. Ejemplo: Chia.

**JEST** es una framework que incluye tanto el test runner como la assertion liberty.

Framework + utilizados:

- JUnit: Herramienta de prueba de uso gratuito que se utiliza para el lenguaje de programación Java. Proporciona afirmaciones para identificar el método de prueba. Esta herramienta prueba los datos primero y luego los inserta en el fragmento de código.
- NUnit: Es un marco de trabajo de pruebas unitarias ampliamente utilizado para todos los lenguajes .net. Es una herramienta de código abierto y admite pruebas basadas en datos que pueden ejecutarse en paralelo.
- JMockit: Es una herramienta de prueba unitaria de código abierto. Es una herramienta de cobertura de código con métricas de sentencia y decisión. Permite hacer mocks de API con sintaxis de grabación y verificación. Esta herramienta ofrece cobertura de sentencia, cobertura de decisión y cobertura de datos.
- EMMA : Es un conjunto de herramientas de código abierto para analizar y reportar código escrito en lenguaje Java. Emma admite tipos de cobertura como método, sentencia, bloque básico. Está basado en Java, por lo que no tiene dependencias de bibliotecas externas y puede acceder al código fuente.
- PHPUnit: Es una herramienta de prueba unitaria para programadores PHP. Toma pequeñas porciones de código que se denominan unidades y prueba cada una de ellas por separado. La herramienta también permite a los desarrolladores usar métodos de confirmación predefinidos para afirmar que un sistema se comporta de cierta manera.

### Test unitario con JavaScript

Vamos a utilizar el framework JEST. Si queremos configurarlo, debemos instalar:

1. NodeJS (<https://nodejs.org/>)
2. Un IDE (el recomendado es Visual Studio Code - <https://code.visualstudio.com/>)
3. JEST (<https://jestjs.io/>)

Crear nuestro primer unit test:

- 1) Bajar el código fuente del siguiente repositorio. Este programa solicita el ingreso de Nombre y Edad y al presionar el botón Agregar Usuario arma una lista con los datos ingresados.
  - 2) Configurar JEST como ejecutor de las pruebas:
    1. Ir al archivo package.json
    2. En "test" que está dentro de "script" ingresar jest
  - 3) Crear la carpeta y archivo de prueba:
    1. Crear la carpeta \_\_test\_\_ donde se encontrarán todos los archivos de prueba
    2. Crear el archivo de prueba util.test.js. Para que JEST reconozca los archivos de prueba deben terminar en .test.js
  - 4) Importar el código a probar: en este caso vamos a probar el que se encuentra en el archivo util.js  
`const { generateText } = require('../util.js');`
  - 5) Escribir el código del unit test. Las funciones claves a utilizar son:
    - test() o it(): Se define el test. Se debe ingresar un nombre representativo.
    - expect(): lo que se desea verificar, es parte de la assertion library. En el ejemplo se quiere verificar la salida que se genera al presionar el botón Agregar Usuario.
  - 6) Para ejecutar el test, primero debemos abrir la terminal y, luego, ejecutar el comando `npm test`.
  - 7) Se puede configurar que los test se ejecuten cada vez que se genera un cambio en el código. Para ello, debemos ir al archivo package.json e ingresar `jest --watchAll` en el nodo "test" que está dentro de "script".  
Luego, ejecutar el siguiente comando: `npm test`  
Allí, se debe realizar alguna modificación en el código del programa (util.js) y por último, al momento de guardar el archivo modificado, se ejecutarán los test.
  - 8) Para generar agrupaciones o suite de unit test se utiliza la siguiente palabra clave: `describe()` → será útil p/ organizar las pruebas según las funcionalidades a probar.
  - 9) Jest usa "matchers" o "comparadores" para permitirle probar valores de diferentes maneras. Algunos de los más relevantes son:
    - `.toBe()`: utiliza `Object.is` para probar la igualdad exacta.
    - `.toEqual()`: para verificar el valor de un objeto.
    - `.toMatch()`: para comparar cadenas con expresiones regulares.
- La lista completa se encuentra en: <https://jestjs.io/docs/expect>

### **Técnicas de prueba de caja blanca**

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

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

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

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

- Prueba y cobertura de sentencia :
  - Ejercita las sentencias ejecutables en el código.



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

$$\text{Cobertura (\%)} = \frac{\text{número de sentencias ejecutadas por las pruebas}}{\text{número total de sentencias ejecutables en el objeto de prueba}} \times 100$$

- Prueba y cobertura de decisión

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

$$\text{Cobertura (\%)} = \frac{\text{número de resultados de decisión ejecutados por las pruebas}}{\text{número total de resultados de decisión en el objeto de prueba}} \times 100$$

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.

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

El desarrollo ágil exige comentarios periódicos para desarrollar el producto esperado. Existe una alta probabilidad de que los requisitos del proyecto cambien durante el ciclo del sprint de desarrollo. Para lidiar con esto, los equipos necesitan retroalimentación constante para evitar distribuir software inutilizable. TDD está diseñado para ofrecer tal retroalimentación desde el principio.

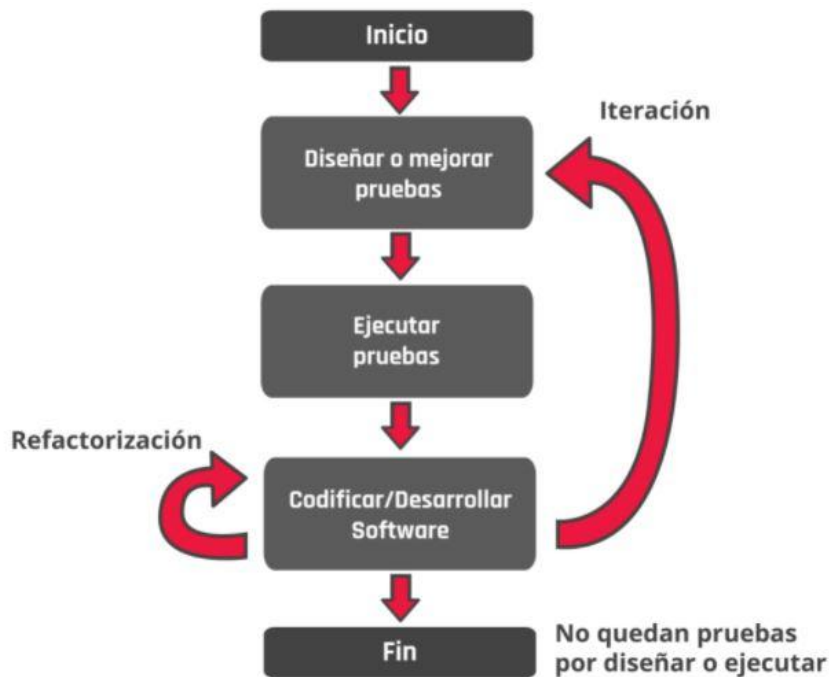
-El enfoque de “prueba primero” de TDD también ayuda a mitigar los cuellos de botella críticos que obstruyen la calidad y la entrega del software. Basado en la retroalimentación, la corrección de errores y la adición de nuevas funciones, el sistema evoluciona para garantizar que todo funcione según lo esperado.

-TDD mejora la colaboración entre los miembros del equipo tanto del desarrollo como de los equipos de control de calidad, así como con el cliente.

-Como las pruebas se crean de antemano, los equipos no necesitan perder tiempo recreando extensos scripts de prueba.

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

### Test Driven Development (TDD)



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

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

TDD es una metodología muy útil, pero no es adoptada por todas las empresas o en todos los proyectos.

- Implica dedicarle más tiempo a cada tarea, lo que a su vez se traduce en tiempo de desarrollo y costos para el empleador.



## Desarrollo guiado por pruebas (TDD)

- ✖ Es un proceso de desarrollo de software donde se desarrolla el código guiado por casos de prueba automatizados.

Las pruebas escritas son principalmente de nivel unitario y se centran en el código, aunque también pueden escribirse pruebas a nivel de integración o de sistema.

Ayuda a los desarrolladores a concentrarse en resultados esperados claramente definidos.

Existe menor redundancia debido a que las pruebas se automatizan y se utilizan en la integración continua.

Mayor calidad en el código desarrollado.

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.

El proceso consiste en:

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

# Vs.

## Desarrollo guiado por el comportamiento (BDD)



Es un proceso de desarrollo de software que permite al desarrollador concentrarse en probar el código basándose en el comportamiento esperado del software.

Las pruebas escritas son principalmente de nivel de sistema e integración, aunque también se pueden utilizar para escribir pruebas unitarias. Las pruebas escritas son principalmente de nivel de sistema e integración, aunque también se pueden utilizar para escribir pruebas unitarias.

Ayuda al desarrollador o probador a colaborar con otras partes interesadas para definir pruebas precisas centradas en las necesidades del negocio.

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 software debido a que todo el equipo puede entender y colaborar en las pruebas.

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.

El proceso consiste en:

1. Se busca un lenguaje común, llamado lenguaje natural, para unir las especificaciones técnicas y los requisitos del cliente / negocio (historias de usuario), generalmente se utiliza Gherkin.
2. Se definen los criterios de aceptación de cada historia de usuario. Pueden utilizarse marcos de desarrollo guiados por el comportamiento (frameworks como Cucumber, Jbehave, Specflow) para definir criterios de aceptación basados en el formato **dado / cuando / entonces** (Given / When / Then).
  - **Dado** un contexto inicial,
  - **cundo** se produce un evento,
  - **entonces**, se aseguran algunos resultados.
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, es decir, se implementa el comportamiento para cada línea en lenguaje natural.
5. Se ejecutan los casos de prueba y se refactoriza.

Definición

Nivel de prueba

Utilidad

Nivel de prueba

Utilidad

Redundancia /

Nivel de prueba

Utilidad

Nivel de prueba

Definición

Nivel de prueba

Nivel de prueba

Utilidad

Redundancia /

Definición

Nivel de prueba

Utilidad

Redundancia /  
Retrabajo

Calidad

Productividad

Proceso

## Mock y Stub

- Mock: objetos preprogramados con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs, aunque sus diferencias son sutiles. En el mock podemos definir expectativas con todo lujo de detalles. Valida el comportamiento en la colaboración.
- Stub: proporcionan respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden grabar información sobre las llamadas.  
El stub tan solo devuelve respuestas preprogramadas a posibles llamadas.  
Simula respuestas a consultas.

## CLASE 17 SEMANA 6

### INTRODUCCIÓN - API TESTING:

¿Qué es el testing de back end?

Una aplicación web típica tiene tres capas: interfaz (UI), lógica empresarial y una base de datos. Probar la interfaz (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

La información es compartida entre diferentes sistemas gracias a **APIs**

**Application Programming Interface** → URL que nos devuelve información para que otro sistema lo consuma. Son desarrolladas para que dos sistemas puedan comunicarse. Las APIs suelen tener su propia documentación.

Hay APIs públicas, privadas y semipúblicas.

- Públicas → gratuitas y podemos consultar su información. Capaz necesitamos registrarnos. RESTCountries por ejemplo.
- Semipúblicas → Registrarnos y seguramente nos ponga un límite. Puedo acceder a determinados tweets por día y si yo quiero pago y Twitter me da canilla libre de todos los Tweets que yo quiera.
- Privadas → Netflix. Utilizan APIs internas para poder comunicarse entre diferentes aplicaciones pero que son privadas. Los endpoints no están disponibles para nuestro consumo, son solo para productos en sí.

**Endpoint** → Cuando hablamos de APIs, hace referencia al punto de conexión donde necesitamos apuntar para obtener la información que queremos. Son las URL que debemos utilizar para obtener información de un servidor a través de una API.

## Testing de APIs

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.

Protocolos de comunicación → Definen las reglas y códigos de comunicaciones necesarios para entendernos. Dentro de un esquema web se rige a través del protocolo HTTP.

**HTTP** → define una serie de lineamientos que se deben cumplir p/ procesar la información.

- Gestiona las transacciones web entre 2 entidades, el cliente y el servidor. Cada vez que se hace un pedido del cliente a un servidor, este pedido viene acompañado de un método o verbo con los que trabaja HTTP.

**Método** → es una función que implementa una serie de procesos lógicos que define lo que sucederá cuando se accede a un recurso.

- GET → Método a través del cual vas a poder solicitar cosas al servidor. Las peticiones hechas con Get solo deben recuperar datos, no modificar información que se encuentra en el servidor. Solamente las rutas que usen el método GET podrán ser accedidas desde la URL o desde un enlace. Pedir resultados enviando lo que necesitamos, ejemplo Rosalía en el buscador.  
\*Los próximos métodos necesitarán ser procesados generalmente en un formulario:
- POST → método a través del cual podrás enviar datos al servidor. En general, servirán para crear registros dentro de tu aplicación.  
Cuando te estás registrando y estás escribiendo los datos que te solicitan, tu petición va a viajar por POST. Cuando estas creando un producto dejando un comentario nuevo o realizando una compra, estas creando un nuevo registro del lado del servidor.
- POST es más seguro para enviar información del cliente al servidor comparado con GET → toda la información que envíes será visible en la URL del navegador, en cambio, cuando realizas un pedido con POST esa información está oculta. Los pedidos por POST no pueden ser cacheados, ni estar en favoritos ni en tu historial del navegador.
- PUT → Se usa para reemplazar información actual de un registro ya presente en el servidor. Cuando editas datos en facebook, o cambias tu biografía en Instagram, tu petición debería estar usando el método PUT → no estas queriendo crear un nuevo registro, sino editar uno ya existente.
- DELETE → Borra un recurso presente en el servidor. Eliminar una foto en facebook. Vas a necesitar información que identifique ese registro que estás por editar o eliminar → se suele utilizar el ID del registro.

## CLASE 19 SEMANA 7

### API TESTING:

**Testing método GET** → recuperar información de una URL específica y analizar la información obtenida a partir de los test.

## POSTMAN

### GET

- 1) Debemos crear una nueva solicitud en Postman. Para realizarlo, se debe hacer clic en la pestaña "New".
- 2) El siguiente paso es crear la solicitud. →
  - a. Configura su solicitud HTTP en GET.
  - b. Ingresa el enlace en la URL de la solicitud  
(<https://jsonplaceholder.typicode.com/users>)
  - c. Haz clic en ENVIAR para mandar la solicitud al servidor que aloja la URL.
- 3) Cuando aparezca el mensaje 200 OK significa que la solicitud se realizó correctamente. **Resultados:**
  - a. Response: es la información en plano devuelta por el servidor. Con esto podemos dar una revisión temprana de los datos de la aplicación.
  - b. 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.
  - c. Código de respuesta: cuando solicitas información al servidor, este puede contestar distintos códigos de estado que te informan qué pasó con tu solicitud. Una API nos devuelve diferentes códigos de respuesta que nos informan qué pasó con la petición. Estas respuestas se agrupan en 4 clases:
    - Códigos 200 → Respuestas exitosas.
    - Códigos 300 → Redirecciones.
    - Códigos 400 → Error del cliente.
    - Códigos 500 → Error del servidor.
  - d. Cookies: nos permiten ver la información relacionada con la sesión.
  - e. Headers: información sobre la solicitud procesada.

**Testing 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.

### POST

- 1) Se debe crear una nueva solicitud en Postman. Hacemos clic en la pestaña "New".
- 2) El siguiente paso, es crear la solicitud. ¿Cómo?
  - a. Configurar su solicitud HTTP en POST.
  - b. Ingresar el enlace en la URL de la solicitud  
(<https://jsonplaceholder.typicode.com/users>).
  - c. 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-urlencoded: se envían los datos como si fuera un formulario.
  - d. Haz clic en el cuerpo de la solicitud y selecciona la opción "raw"(a), luego "Json"(b). Copia y pega lo brindado en la diapositiva anterior en el body (c).
  - e. Haz clic en ENVIAR para mandar la solicitud al servidor que aloja la URL.
- 3) Si aparece el mensaje 201 CREATED, significa que la solicitud se realizó correctamente.

## API's Testing - Automatización de pruebas

### *¿Qué son las pruebas automatizadas?*

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: 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 un 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 javascript con algunas pruebas por defecto, pero también somos libres de escribir nuestras propias pruebas utilizando javascript.

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

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:

```
pm.test("Descripción Funcionalidad a Probar", function(){  
    // Código que valida la prueba del test  
});
```

- 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.

```
pm.test("Obteniendo contenido de la Response", function(){  
    pm.response.json();  
});
```

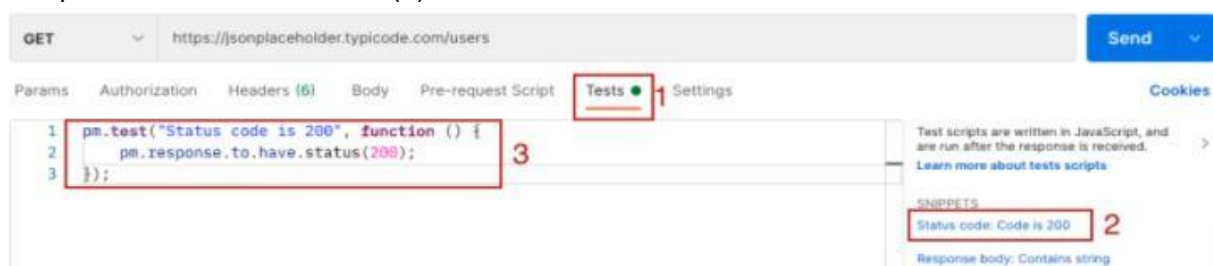
- Otro método importante es el que nos permite realizar una comprobación de contenido, este es pm.expect.



```
pm.test("Comparando el valor devuelto con el esperado", function(){
    pm.expect(valor).to.equal("Valor esperado");
});
```

2. 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:

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

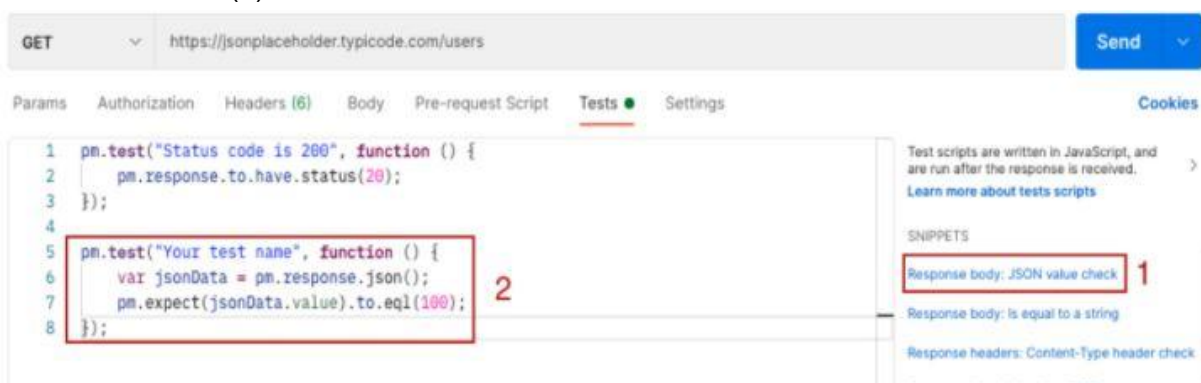


- Al hacer clic en ENVIAR se mostrará el resultado del test.
- Con esta prueba estamos validando que el código de respuesta de la API sea 200. Si esto es correcto el test devolverá PASS: eso significa que el servicio está respondiendo según lo esperado.
- Si el servicio falla nos mostrará el estado FAIL, 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 javascript.

3. 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.

```
pm.test("Verificar si Leanne Graham tiene el ID de usuario 1", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData[0].name).to.eql("Leanne Graham");
});
```

- 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** → Son un grupo de peticiones guardadas que pueden organizarse en carpetas. Esto nos permite agrupar y administrar nuestras peticiones de manera + eficiente.

- 1) 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?
  - a. Ingresa el nombre de la colección.
  - b. Ingresa una descripción para la misma (opcional).
  - c. Haz clic en CREAR para crear la nueva colección.
- 3) Puedes agregar cualquier número de peticiones a una colección. Solo debes arrastrar la petición a la carpeta de la colección.

**Variables de entorno** → Solemos utilizar la misma solicitud varias veces con datos diferentes. Postman nos permite parametrizar estos datos y guardarlos en forma de archivo o en variables de entorno. Una variable de entorno se guarda en el entorno de trabajo. Estas se pueden crear de manera estática o dinámica.

Podríamos tener diferentes entornos para Dev, QA y Producción, con sus respectivas variables.

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

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

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

Cuestionario:

1. El método POST nos sirve cuando necesitamos agregar datos a nuestra aplicación. Enviamos estos datos a través de → Body → viajan más seguros.
2. El código de respuesta 200 de un API nos indica un error en la petición → Falso
3. ¿Es este test correcto para validar si el contenido devuelto es igual al esperado?

```
pm.test("Nombre es Víctor", function() {  
    nombre = pm.response.json().nombre;  
    pm.expect(nombre).to.equal("Víctor");  
});
```

Verdadero → Con este test obtenemos el nombre del objeto devuelto en la petición y validamos si es igual al nombre esperado.

4. Postman Test nos permite ejecutar scripts con Java para validar si nuestras APIs funcionan como se esperaba. → Verdadero → Utiliza JavaScript como lenguaje para crear scripts.
5. Las colecciones nos permiten guardar peticiones en grupos para tenerlas organizadas en carpetas y administrarlas de manera más eficiente. → Verdadero

## CLASE 20 SEMANA 7

### 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.

Se pueden automatizar los siguientes tipos de pruebas:

- Pruebas unitarias.
- Pruebas de API.
- Pruebas de interfaz gráfica.
- Pruebas de rendimiento, pruebas de regresión, de integración, de seguridad, pruebas de compatibilidad en diferentes navegadores, casos repetitivos

### Objetivos de automatización:

- Mejora 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	Desventajas
<ul style="list-style-type: none"><li>• Pruebas complejas.</li><li>• Ahorro de tiempo en la ejecución de pruebas.</li><li>• Las pruebas están menos sujetas a errores del <i>tester</i>.</li><li>• Se pueden rehusar los scripts con pruebas automatizadas.</li></ul>	<ul style="list-style-type: none"><li>• El equipo necesita tener conocimientos técnicos para poder implementar los scripts de test.</li><li>• Requiere un mantenimiento continuo.</li><li>• No podemos automatizar aspectos como el nivel de usabilidad.</li><li>• Requiere tecnologías adicionales.</li></ul>

### **Patrones de diseño**

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

### **Patrones de diseño en automatización**

- Screenplay:
  - Este patrón tiene un enfoque de desarrollo encaminado por el comportamiento Behaviour Driven Development (BDD). Es una estrategia de desarrollo que se enfoca en prevenir defectos en lugar de encontrarlos en un ambiente controlado.
  - Presenta un alto desacoplamiento de la interfaz de usuario.
  - Propone trabajar en términos de tareas y no de interfaz de usuario.
  - Cumple con los principios SOLID.
- Page Object:
  - Es un patrón de diseño que representa los componentes web o página web en una clase.
  - Se utiliza en las pruebas automatizadas para evitar código duplicado y mejorar el mantenimiento de las mismas.
  - No cumple con los principios SOLID.

### **¿Qué son los principios SOLID?**

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

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

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

En definitiva, desarrollar un **software de calidad**.

SOLID es un acrónimo para definir los 5 principios básicos de la programación orientada a objetos:

- + Single responsibility → Establece que una clase, componente o microservicio debe ser responsable de una sola cosa ( decoupled en inglés). Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra.
- + Open-closed → Establece que las entidades software (clases, módulos y funciones) deberían estar abiertas para su extensión, pero cerradas para su modificación.
- + Liskov substitution → Declara que una subclase debe ser sustituible por su superclase. Si al hacer esto el programa falla estaremos violando este principio. Cumpliendo con esto se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.
- + Interface segregation → Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.  
Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este no usa, pero que otros clientes sí lo hacen, este estará siendo afectado por los cambios que fueren los demás en dicha interfaz.
- + Dependency inversion → Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:
  - Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
  - Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.En algún momento nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase debemos usar inyección de dependencias: nos permitirá controlar las funcionalidades desde un sitio concreto en vez de tenerlas esparcidas por todo el programa.

SOLID tiene bastante relación con los patrones de diseño.

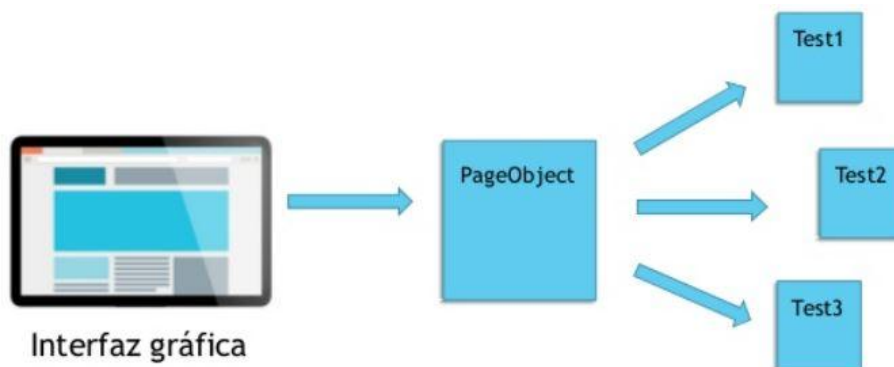
## Page Object Model

POM → Es la implementación del patrón de diseño Page Object utilizado en la automatización de pruebas. Tiene como objetivo mejorar el mantenimiento de las pruebas y reducir la duplicación de código.

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

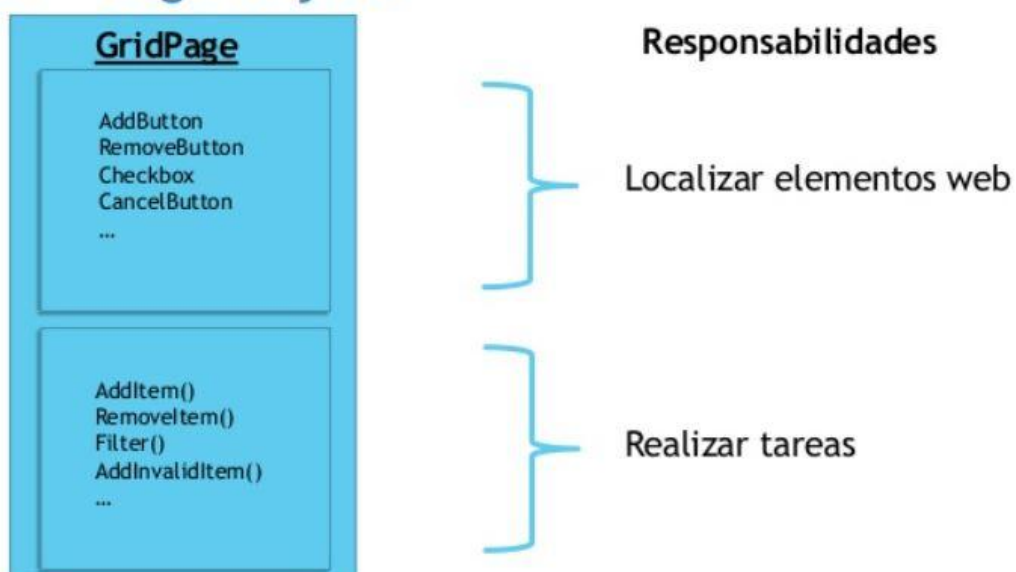
Las páginas web se representan como clases y los diversos elementos de la página se definen como variables en la clase. Todas las interacciones de usuario posibles se pueden implementar como métodos en la clase.

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



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

## SRP - Page Object



## 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.

→ **Selenium** es una herramienta de código abierto para la automatización de pruebas de navegadores web. Proporciona la posibilidad de grabar y/o reproducir, editar y depurar casos de pruebas que permitirán ejecutarlas repetidamente cuando sea necesario.

Selenium ofrece tres productos con distintos propósitos:

- **Selenium Grid** → Permite diseñar pruebas automatizadas para aplicaciones web en diversas plataformas. Asimismo, posibilita la ejecución de pruebas en diversos servidores en paralelo. Es por esto por lo que reduce el tiempo de ejecución y el costo, debido a la ejecución de las pruebas en varios navegadores y en diversos S.O. Selenium Grid cuenta con dos componentes: Selenium Hub y Remote Control.
- **Selenium WebDriver** → Es una herramienta que permite automatizar pruebas UI (User Interface) o interfaz de usuario de aplicaciones web. Algunos de los lenguajes que son soportados son Java, C#, Python, Ruby, PHP, JavaScript. Proporciona APIs orientadas a objetos en una variedad de idiomas. Esto nos permite tener un mayor control sobre la aplicación de las prácticas de desarrollo de software estándar. Es útil para poder simular la manera en que los usuarios reales interactúan con alguna aplicación web.
- **Selenium IDE** → (Integrated Development Environment - Entorno de Desarrollo Integrado) → este componente es una herramienta de automatización que nos permite grabar, editar y depurar pruebas. Tb se lo conoce → Selenium Recorder.

## Selenium IDE:

Se trata de un entorno de pruebas de software para aplicaciones basadas en la web que permite realizar tareas de Record&Play de flujos de pruebas.

Los flujos grabados quedan contenidos en un script que se puede editar y parametrizar para adaptarse a los diferentes casos, y lo que es más importante, su ejecución se puede repetir tantas veces como se quiera.

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

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

Asimismo, las acciones pueden ser ejecutadas paso a paso.

## Record&Play

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".
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.



En esta grilla se visualizan:

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

Cuestionario:

1. ¿Qué es Selenium? → Es una herramienta de código abierto para la automatización de pruebas de navegadores web. Selenium es una de las suites de pruebas automatizadas más usadas open source que fue diseñada para soportar pruebas de automatización. Tiene compatibilidad con varios navegadores y plataformas y soporta diversos lenguajes de programación.
2. Selenium soporta pruebas funcionales y de regresión → V.
3. ¿Cuál es la función principal de Selenium IDE? → Su función principal es grabar/reproducir casos de pruebas. Registra las acciones de los usuarios en el navegador, utilizando comandos de Selenium existentes.
4. ¿Cuáles son los diferentes tipos de localizadores que soporta Selenium? → ID, Name, LinkText, Xpath, y CSS Selector.

## CLASE 22 SEMANA 8

### AUTOMATIZACIÓN DE LA PRUEBA:

#### Selenium WebDriver

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

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

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

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

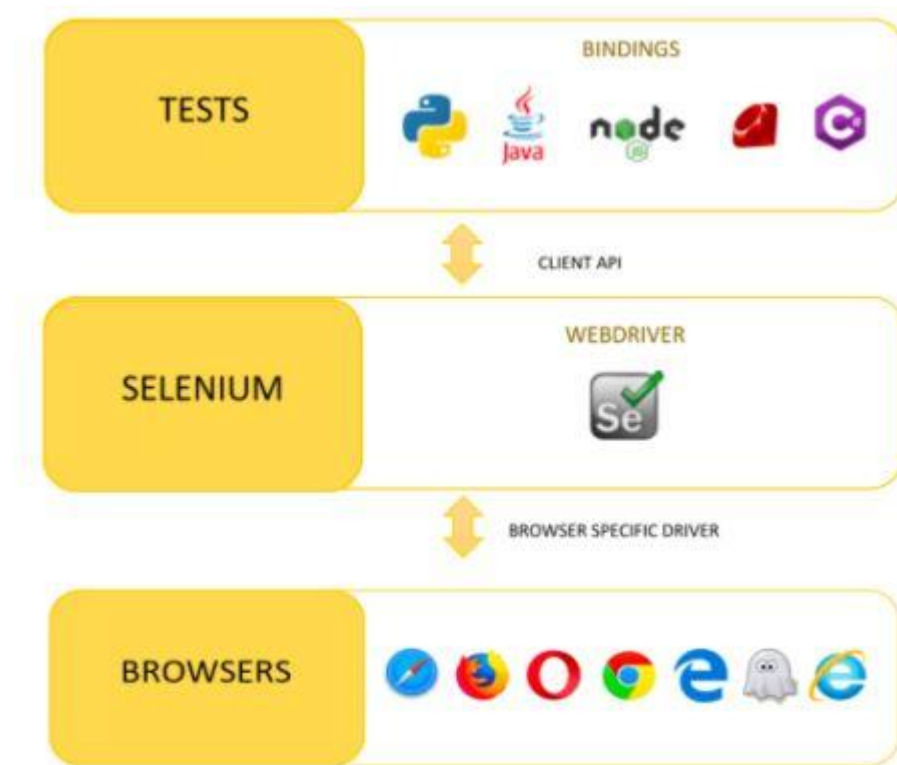
Selenium es compatible con muchos lenguajes de programación como Java, C #, Python, etc., y con múltiples navegadores como Google Chrome, Firefox, Internet Explorer, etc.

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

- Los tests a ejecutarse (escritos con un lenguaje a elección entre varios como Java, C#, Node JS, etc); "Client API".
- Un servidor standalone en donde se ejecutarán los casos de prueba.

- Un browser driver que conectará los scripts generados con la client
- API con el browser a través del Selenium Server.

Gráficamente puede verse así:



### Instalación de Selenium WebDriver

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

#### 1. Instalando las librerías de Selenium

Primero debes instalar las librerías de Selenium para tu proyecto de automatización. El proceso de instalación depende del lenguaje que elijas usar. En este caso te pediremos que instales las librerías de JavaScript → se puede hacer usando npm:

**npm install selenium-webdriver**

#### 2. Instalando los binarios de WebDriver

Para ejecutar tu proyecto y controlar el navegador debes tener instalados los binarios de WebDriver específicos para el navegador.

En este caso te pediremos que instales chromedriver.

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

Una vez instalados todos los componentes, vamos a crear nuestro primer script de prueba.

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