

## ¿Qué es testing?

Es un proceso que implica una serie de actividades, buscamos planificar, ejecutar, trabajar con lo que es el avance e implementacion de las pruebas y obviamente la puesta en practica.

Comprobamos resultados en funcion a lo que nosotros como probadores de software estamos buscando detectar. Si tenemos discrepancia trabajamos con defectos.

- Ejecucion de prueba y comprobacion de resultados.
- Planificacion de las pruebas
- Analisis, diseño e implementacion de las pruebas
- Notificacion del avance y resultado de la ejecucion de pruebas
- Evaluacion de la calidad de un objeto de prueba.

Caso de uso es user story (requerimientos), caso de prueba es el documento que indica como se va a testear, diseñado por test manager o equipo de testing y ejecutado por testers generalmente, excepto aquellos que requieran un conocimiento especializado y tecnico como por ejemplo relacionados con el codigo.

### Error defecto falla

Error es humano // mal interpretacion de un requisito

defecto es el resultado de un error humano // falta de funcionalidades

falla el sistema a causa de un/os defecto // fallo global en el sistema, ej que no se puedan conectar, es decir que no se pueda hacer nada.

## 7 Principios de testing

1. La prueba muestra la presencia de defectos, no su ausencia
2. La prueba exhaustiva es imposible
3. La prueba temprano ahorra tiempo y dinero
4. Los defectos se agrupan
5. Cuidado con la prueba del pesticida
6. La prueba se realiza de manera diferente según el contexto
7. La ausencia de errores es una falacia.

### Ciclo de vida:

1. Inicial/nuevo: se recopila la informacion y se registra el defecto
2. Asignado: se analiza y trabaja en la solucion
3. En proceso: se analiza y trabaja en la solucion.
4. Corregido: se realizan los cambios de codigo para solucionar el defecto

5. En espera de verificación: en espera de que sea asignado a un probador. Es desarrollador esta a la expectativa del resultado de la verificación.
6. En verificación: El probador ejecuta una prueba de confirmación.
7. Verificado: se obtiene el resultado esperado en la prueba de confirmación.
8. Cerrado: el defecto fue corregido y se encuentra disponible para el usuario final.

Participación de usuario/cliente: Solo al inicio y al final.

**Gestión de defectos:** trabajar con lo que es el tratamiento de los defectos. No existe la ausencia de defectos, con lo cual ¿Qué hacemos con ellos? Trabajarlos, resolverlos, priorizarlos. Estableciendo prioridades.

### **Proceso de gestión de defectos:**

1. Detectar (naturalización del defecto).
2. Registrar, varía según su contexto, del componente o sistema, nivel de prueba y modelo de desarrollo elegido.
3. Investigación y seguimiento: me pongo en el lugar del usuario y veo de donde viene el defecto, que fue lo que falló, tomo acciones y saco conclusiones. ¡Analizar!
4. Clasificación/resolución: no podemos clasificar lo que antes no analizamos, una vez termina el paso 3, podemos pasar a la fase de resolución (por parte del equipo de desarrollo). la tarea del tester es probar que los defectos existen, no resolverlos.

Objetivos:

- 1-brindar información sobre cualquier evento adverso, para identificar efectos específicos, aislar el problema con una prueba de producción mínima y corregir los defectos potenciales.
- 2-Proporcionar a los jefes de prueba un medio para hacer seguimiento a la calidad del producto y del impacto de prueba.
- 3-Dar ideas para la mejora de los procesos de desarrollo y prueba.

## Informe de defectos:

Es un registro de salida. Toda la informacion que se va reportando, va a ser la principal entrada para generar el **documento llamado informe de defectos**.

**¿Cómo escribir un buen informe?** Si el defecto se reporta eficientemente las probabilidades de que sea solucionado rapidamente es mayor. Entonces, la solucion de un defecto dependera de la eficiencia con que se reporte.

**Condiciones a tener en cuenta:** Los bugs deben tener identificadores unicos, Una falla debe ser reproducible para reportarla. Ser especifico, Reportar cada paso realizado para reproducirlo.

### Partes de un informe de defectos

- ID
- Titulo
- Descripcion
- Resultado actual
- Resultado esperado
- Pasos para reproduccion
- Estado
- Prioridad
- Severidad
- Reportado por
- Asignado A

### Problemas mas comunes con los informes:

- Redactar de manera excesivamente coloquial y ambigua – dar solo captura de pantalla sin nada mas
- no incluir descripcion del resultado esperado para los pasos realizados
- no determinar un patron con el cual el defecto ocurre antes de reportar el mismo
- no leer defecto reportado siguiendo los pasos un mismo para ver que la descripcion era clara.

¿Qué es testing? La ejecucion de la prueba es solo uno de los pasos en el ciclo de vida de pruebas. Testing es un proceso, conjunto de actividades para encontrar defectos y posteriormente solucionarlos.

Actividades:

- Ejecucion de prueba y comprobacion de resultados
- Planificar las pruebas
- Analizar, diseñar e implementar las pruebas
- Informar el avance y resultado de la ejecucion de pruebas
- Evaluar la calidad de un objeto de prueba

### Principios de testing:

1. La prueba muestra la presencia de defectos, no su ausencia
2. La prueba exhaustiva se imposible

3. La prueba temprana ahorra tiempo y dinero
4. Los defectos se agrupan
5. Cuidado con la prueba del pesticida
6. La prueba se realiza de manera diferente según el contexto
7. La ausencia de errores es una falacia.

Mesa de 3 patas: business analyst, software developer, quality assurance.

## Casos de prueba

Documento escrito que proporciona información escrita sobre qué y cómo probar.

Proceso documentado que define qué se está probando y cómo.

El que es relacionado con los requisitos del usuario y el cómo van a ser los pasos para ejecutar ese caso de prueba.

Que tener en cuenta a la hora de escribir un buen caso de prueba:

1. **Identificador / ID**, la mayoría de herramientas lo generan automáticamente
2. **Nombre del caso de prueba (conciso)**, utilizar una nomenclatura que esté definida, de no existir se recomienda incluir el nombre de módulo al que corresponde el caso de prueba.
3. **Descripción**, debe decir qué se va a probar, ambiente, y datos necesarios para ejecutarlo.
4. **Precondición**, asunción que debe cumplirse antes de ejecutar el caso de pruebas
5. **Pasos**, acciones que se deben realizar para obtener los resultados
6. **Resultados esperados**, lo que se indica al probar cuál debería ser la experiencia luego de ejecutar los pasos y determinar si el test falló o no.
7. **Resultado actual**, resultado al que llegamos GLOBAL, lo que queríamos conseguir en general, no el paso a paso
8. **Estado**, estado del caso de prueba.

No es lo mismo diseñar y ejecutar el caso de prueba, acá hablamos de diseño, no de ejecutar casos de prueba.

Estados de diseño: Durante fase de diseño de prueba.

- **En diseño**
- **Diseñado** // ya se creó y está listo para pasar a un estado de revisión
- **En revisión** // responsabilidad de un test manager o persona en detalle de la operativa en el sector de testing
- **Revisado** // listo para pasar a ejecución

Estados de ejecución: durante fase de ejecución de pruebas.

- **No corrido** // no fue ejecutado
- **Pasado** // pasado correctamente
- **Fallado** // resultado actual discrepante del esperado
- **No aplica** // puede ocurrir por ej: ejecutamos ciertas pruebas pero esas pruebas son de proyectos anteriores o similares, con lo cual para este no aplica y se deja de lado, pero al no correrlo no tiene sentido o sería incorrecto describirlo como fallado

- **Bloqueado** // Tenemos dependencia tal de una condicion que no se puede cumplir al 100% con lo cual no se puede seguir con la ejecucion
- **No completo** // esta en curso, por ejemplo termina la jornada laboral y no terminamos un caso de prueba en curso, la dejamos y queda como no completo hasta retomarla.
- **Diferido** // postergamos su ejecucion, puede que un caso de prueba no sea urgente y se deje para otro sprint.

## Test plan

Es aquel documento donde documentamos nuestros casos de prueba, documentamos posibles estados. El responsable de documentar esto sera el equipo de testing y tal vez un tester manager sera el responsable de revisar el documento.

### Caracteristicas para un buen caso de prueba.

- **Deben ser simples:** debe ser lo mas simple posible ya que otra persona diferente al autor debe poder ejecutarlo, usar lenguaje asertivo para facilitar comprension.
- **El titulo debe ser fuerte:** el titulo debe dar una comprension rapida y completa del caso de prueba
- **Tener en cuenta al usuario final:** debemos ser empaticos y crear casos de prueba que cumplan con los requisitos del cliente, brindando facilidad de uso.
- **No asumir:** no asumir funcionalidad y caracteristicas de la aplicacion, referirse a documentos de especificacion y ante cualquier duda consultar.
- **Asegurar mayor cobertura posible:** escribir casos de prueba para todos los requisitos especificados.
- **Autonomia:** el caso de prueba debe generar lo mismos resultados siempre, sin importar el sujeto que lo pruebe.
- **Evitar la repeticion de casos de prueba:** si se necesita un caso de prueba par ejectuar otro, indicar el caso de prueba por su id.

### Testing positivo y Testing negativo:

**El testing positivo:** trabajamos el flujo normal, lo que esperamos que el software haga a travez de acciones que la persona que ejecuta la prueba pueda identificar.

Ejemeplo: en un login ingresamos usuario y contraseña correcto, se espera que se pueda pasar a la pagina principal. (con esto ya seria un testing positivo)

**El testing negativo:** buscamos un flujo anormal, por ejemplo buscamos mediante parametros incorrectos, siga un funcionamiento no esperado.

Ejemeplo: en un login ingresamos usuario y contraseña incorrecto, se espera que NO se pueda pasar a la pagina principal como si el usuario y contraseña fuesen correctos. Si nos dejase pasar estamos en presencia de un defecto.


**Happy path pertenece al testing positivo:** es un camino de prueba cuidadosamente seleccionado para que una persona puede validar cierta funcionalidad.

Comunmente se utiliza en proyectos "incendiados" para poder demostrar que aunque hay problemas, ciertas funcionalidad son efectivas.

## Ciclo de vida de un defecto

No existe un proceso de prueba unico y universal, pero existen actividades de prueba comunes que nos ayudan a organizarnos para alcanzar los objetivos establecidos.  
El proceso que gestiona un defecto desde su descubrimiento hasta su solucion se denomina ciclo de vida de un defecto.

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

<b>Cerrado</b>  El defecto fue corregido y se encuentra disponible para el usuario final.	<b>1) Nuevo/Inicial</b>  Se recopila la informacion y se registra el defecto Documento: Test Plan.	<b>Asignado</b>  De ser un defecto validado y debe solucionarse asigna al equipo de desarrollo, sino se puede rechazar o diferer. (en casos como duplicado, diferido, devuelto o rechazado)
<b>Verificado</b>  Se obtiene el resultado esperado en la prueba de confirmacion		<b>El proceso</b>  Se analiza y trabaja en la solucion
<b>En verificacion</b>  El probador ejecuta una prueba de confirmacion. Puede reabrirlo, indicando que el defecto no se ha solucionado.	<b>En espera de verificacion</b>  Espera que sea asignado a un probador. El desarrollador esta a la expectativa del resultado de la verificacion	<b>Corregido</b>  Se realizan los cambios en el codigo para solucionar el defecto

## Proceso de gestion de defectos.

- 1) Detectar
- 2) Registrar (varia según el contexto del componente/sistema/nivel de prueba/modelo de desarrollo elegido.
- 3) Investigacion y seguimiento
- 4) Clasificacion/Resolucion

## Objetivos

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

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

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

## Proceso de prueba en contexto:

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

## Tareas principales.

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

## Planificación

Definen objetivos y enfoque de la prueba dentro de restricciones impuestas por contexto.

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

Documentos de salida:

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

## Seguimiento y control.

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

## Subactividades:

- Comprobar resultados y registros de la prueba en relación con los criterios de cobertura especificados.

- Determinar si se necesitan mas pruebas dependiendo del nivel de cobertura que se debe alcanzar.

**Documento de salida**

- Informe de avance de la prueba.

## Analisis

Determina “que probar”

**Subactividades:**

- Analizar base de prueba correspondiente al nivel de prueba considerado. - informacion de diseño e implementacion, la implementacion del componente o sistema en si, informes de analisis de riesgo, etc.-
- Identificar defectos de distintos tipos en las bases de prueba – ambigüedades, omisiones, inconsistencias, inexactitudes, etc.-
- Identificar los requisitos que se van a probar y definir las condiciones de prueba para cada requisito
- Capturar la trazabilidad entre la base de prueba y las condiciones de prueba.

**Documento de salida**

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

## Diseño

Se determina “como probar”

**Subactividades**

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

**Documento de salida:**

- Casos de prueba de alto nivel diseñados y priorizados.

## Implementacion

Se completan los productos de prueba necesarios para la ejecucion de la prueba, incluyendo la secuenciacion de los casos de prueba en procedimientos de prueba.

**Subactividades:**

- Desarrollar y priorizar procedimientos de prueba
- Crear juegos de prueba (test suite) a partir de los procedimientos de prueba
- Organizar los juegos de prueba dentro de un calendario de ejecucion
- Construir el entorno de prueba y verificar que se haya configurado correctamente todo lo necesario.
- Preparar los datos de prueba, los procedimientos de prueba y los juegos de prueba.

**Documento de salida:**

- Procedimientos y datos de prueba
- Calendario de ejecucion
- Test suite.



## Ejecucion

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

### Subactividades:

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

### Documento de salida:

- Reporte de defectos.
- Informe de ejecucion de prueba

## Conclusion

### Subactividades

- Comprobar que todos los informes de defecto estan cerrados.
- Finalizar, archivar y almacenar el entorno de prueba, los datos de prueba, la infraestructura de prueba y otros productos de la prueba para su posterior reutilizacion.
- Transpaso de los productos de prueba a otros equipos que podrian beneficiarse con su uso
- Analizar las lecciones aprendidas de las actividades de prueba completadas.
- Utilizar la informacion recopilada para mejorar la madurez del proceso de prueba.

### Documento de salida:

- Informe resumen de prueba.
- Lecciones aprendidas.

## Niveles de pruebas

Grupo de actividades que estan integradas, pero no se mezclan con los tipos de prueba.

Pruebas llevadas a cabo por los desarrolladores, necesita personas con concimiento tal que excede a los testers. Un ejemplo de componente es un boton.  
prueba de componente también llamada prueba unitaria es por definición la prueba que se lleva a cabo luego de haber construido el componente.

Para toda prueba de componente hay que tener en cuenta: (a) las bases de la prueba, es decir: los requisitos del componente, su diseño detallado y su código; (b) los objetos de prueba más típicos, es decir: los componentes, ó clases, ó unidades, ó módulos, más programas y/o conversión de datos y/o migración de programas; y finalmente (c) los módulos de la/s base/s de datos.

UNA UNIDAD DE SOFTWARE ESPECIFICA, UN EVENTO. El login, un boton.

### Prueba unitaria o 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.

#### Bases de prueba

Algunos ejemplos de productos de trabajo que se pueden utilizar como base de prueba incluyen:

- Diseño detallado.
- Código.
- Modelo de datos.
- Especificaciones de los componentes.

#### Objeto de prueba

Los objetos de prueba característicos para la prueba de componente incluyen:

- Componentes, unidades o módulos.
- Código y estructuras de datos.
- Clases.
- Módulos de base de datos.

### Prueba unitaria o de componente

#### Defectos y fallos característicos

Ejemplos de defectos y fallos característicos de la prueba de componente incluyen:

- Funcionamiento incorrecto –por ejemplo, no lo hace de la manera en que se describe en las especificaciones de diseño—.
- Problemas de flujo de datos.
- Código y lógica incorrectos.

#### Enfoques y responsabilidades específicas

En general, el desarrollador que escribió el código realiza la prueba de componente. Los desarrolladores pueden alternar el desarrollo de componentes con la búsqueda y corrección de defectos. A menudo, estos escriben y ejecutan pruebas después de haber escrito el código de un componente. Sin embargo, especialmente en el desarrollo ágil, la redacción de casos de prueba de componente automatizados puede preceder a la redacción del código de la aplicación.

Las pruebas de integración dentro del software testing chequean la integración o interfaces entre componentes, interacciones con diferentes partes del sistema, como un sistema operativo, sistema de archivos y hardware o interfaces entre sistemas. Las pruebas de integración son un aspecto clave del software testing.

Integración entre 2 componentes, por ejemplo primero alta de usuario y después login.

## Prueba de integración

Objetivos específicos	Bases de prueba	Objeto de prueba
<p>La prueba de integración se centra en las interacciones entre componentes o sistemas.</p> <ul style="list-style-type: none"> <li>• Reducir el riesgo.</li> <li>• Verificar que los comportamientos funcionales y no funcionales de las interfaces sean los diseñados y especificados.</li> <li>• Generar confianza en la calidad de las interfaces.</li> <li>• Encontrar defectos –que pueden estar en las propias interfaces o dentro de los componentes o sistemas–.</li> <li>• Prevenir la propagación de defectos a niveles de prueba superiores.</li> </ul>	<p>Algunos ejemplos de productos de trabajo que pueden utilizarse como base de prueba incluyen:</p> <ul style="list-style-type: none"> <li>• Diseño de software y sistemas.</li> <li>• Diagramas de secuencia.</li> <li>• Especificaciones de interfaz y protocolos de comunicación.</li> <li>• Casos de uso.</li> <li>• Arquitectura a nivel de componente o de sistema.</li> <li>• Flujos de trabajo.</li> <li>• Definiciones de interfaces externas.</li> </ul>	<p>Los objetos de prueba característicos para la prueba de integración incluyen:</p> <ul style="list-style-type: none"> <li>• Subsistemas.</li> <li>• Bases de datos.</li> <li>• Infraestructura.</li> <li>• Interfaces.</li> <li>• Interfaces de programación de aplicaciones –API por sus siglas en inglés–.</li> <li>• Microservicios.</li> </ul> 

## Prueba de integración

Defectos y fallos característicos	Enfoques y responsabilidades específicas
<ul style="list-style-type: none"> <li>• Datos incorrectos, datos faltantes o codificación incorrecta de datos.</li> <li>• Secuenciación o sincronización incorrecta de las llamadas a la interfaz.</li> <li>• Incompatibilidad de la interfaz.</li> <li>• Fallos en la comunicación entre componentes.</li> <li>• Fallos de comunicación entre componentes no tratados o tratados de forma incorrecta.</li> <li>• Suposiciones incorrectas sobre el significado, las unidades o las fronteras de los datos que se transmiten entre componentes.</li> </ul>	<p>La prueba de integración debe concentrarse en la integración propiamente dicha. Se puede utilizar los tipos de prueba funcional, no funcional y estructural. En general es responsabilidad de los testers.</p> 



Ejemplo de prueba de sistema: alta de usuario, verificar las pruebas de zoom, ver todo el material de cada clase, ver el progreso que tengo en una clase.  
Es trabajo de un tester, es diferente de la prueba de aceptación en que la realiza un tester y no el usuario final.

Prueba de sistema		
Objetivos específicos	Bases de prueba	Objeto de prueba
<ul style="list-style-type: none"> <li>• Reducir el riesgo.</li> <li>• Verificar que los comportamientos funcionales y no funcionales del sistema son los diseñados y especificados.</li> <li>• Validar que el sistema está completo y que funcionará como se espera.</li> <li>• Generar confianza en la calidad del sistema considerado como un todo.</li> <li>• Encontrar defectos.</li> <li>• Prevenir la propagación de defectos a niveles de prueba superiores o a producción.</li> </ul>	<p>Algunos ejemplos de productos de trabajo que se pueden utilizar como base de prueba incluyen:</p> <ul style="list-style-type: none"> <li>• Especificaciones de requisitos del sistema y del software –funcionales y no funcionales–.</li> <li>• Informes de análisis de riesgo.</li> <li>• Casos de uso.</li> <li>• Épicas e historias de usuario.</li> <li>• Modelos de comportamiento del sistema.</li> <li>• Diagramas de estado.</li> <li>• Manuales del sistema y del usuario.</li> </ul>	<ul style="list-style-type: none"> <li>• Aplicaciones.</li> <li>• Sistemas hardware/software.</li> <li>• Sistemas operativos.</li> <li>• Sistema sujeto a prueba (SSP).</li> <li>• Configuración del sistema y datos de configuración.</li> </ul> 

Prueba de sistema	
Defectos y fallos característicos	Enfoques y responsabilidades específicas
<ul style="list-style-type: none"> <li>• Cálculos incorrectos.</li> <li>• Comportamiento funcional o no funcional del sistema incorrecto o inesperado.</li> <li>• Control y/o flujos de datos incorrectos dentro del sistema.</li> <li>• Incapacidad para llevar a cabo, de forma adecuada y completa, las tareas funcionales extremo a extremo.</li> <li>• Fallo del sistema para operar correctamente en el/los entorno/s de producción.</li> <li>• Fallo del sistema para funcionar como se describe en los manuales del sistema y de usuario.</li> </ul>	<p>La prueba de sistema debe centrarse en el comportamiento global y extremo a extremo del sistema en su conjunto, tanto funcional como no funcional. Deben 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.</p> 

Es similar a la prueba de sistema, pero es llevada a cabo por el usuario o cliente.

Prueba de aceptación		
Objetivos específicos	Bases de prueba	Objeto de prueba
<p>La prueba de aceptación, al igual que la prueba de sistema, se centra normalmente en el comportamiento y las capacidades de todo un sistema o producto. Los objetivos de la prueba de aceptación incluyen:</p> <ul style="list-style-type: none"> <li>• Establecer confianza en la calidad del sistema en su conjunto.</li> <li>• Validar que el sistema está completo y que funcionará como se espera.</li> <li>• Verificar que los comportamientos funcionales y no funcionales del sistema sean los especificados.</li> </ul>	<p>Entre los ejemplos de productos de trabajo que se pueden utilizar como base de prueba se encuentran:</p> <ul style="list-style-type: none"> <li>• Procesos de negocio.</li> <li>• Requisitos de usuario o de negocio.</li> <li>• Normativas, contratos legales y estándares.</li> <li>• Casos de uso.</li> <li>• Requisitos de sistema.</li> <li>• Documentación del sistema o del usuario.</li> <li>• Procedimientos de instalación.</li> <li>• Informes de análisis de riesgo.</li> </ul>	<ul style="list-style-type: none"> <li>• Sistema sujeto a prueba.</li> <li>• Configuración del sistema y datos de configuración.</li> <li>• Procesos de negocio para un sistema totalmente integrado.</li> <li>• Sistemas de recuperación y sitios críticos –para pruebas de continuidad del negocio y recuperación de desastres–.</li> <li>• Procesos operativos y de mantenimiento.</li> <li>• Formularios.</li> <li>• Informes.</li> <li>• Datos de producción existentes y transformados.</li> </ul>

Prueba de aceptación	
Defectos y fallos característicos	Enfoques y responsabilidades específicas
<p>Entre los ejemplos de defectos característicos de cualquier forma de prueba de aceptación se encuentran:</p> <ul style="list-style-type: none"> <li>• Los flujos de trabajo del sistema no cumplen con los requisitos de negocio o de usuario.</li> <li>• Las reglas de negocio no se implementan de forma correcta.</li> <li>• El sistema no satisface los requisitos contractuales o reglamentarios.</li> <li>• Fallos no funcionales tales como vulnerabilidades de seguridad, eficiencia de rendimiento inadecuada bajo cargas elevadas o funcionamiento inadecuado en una plataforma soportada.</li> </ul>	<p>A menudo es responsabilidad de los clientes, usuarios de negocio, propietarios de producto u operadores de un sistema, y otros implicados también pueden estar involucrados. La prueba de aceptación se considera, a menudo, como el último nivel de prueba en un ciclo de vida de desarrollo secuencial.</p>

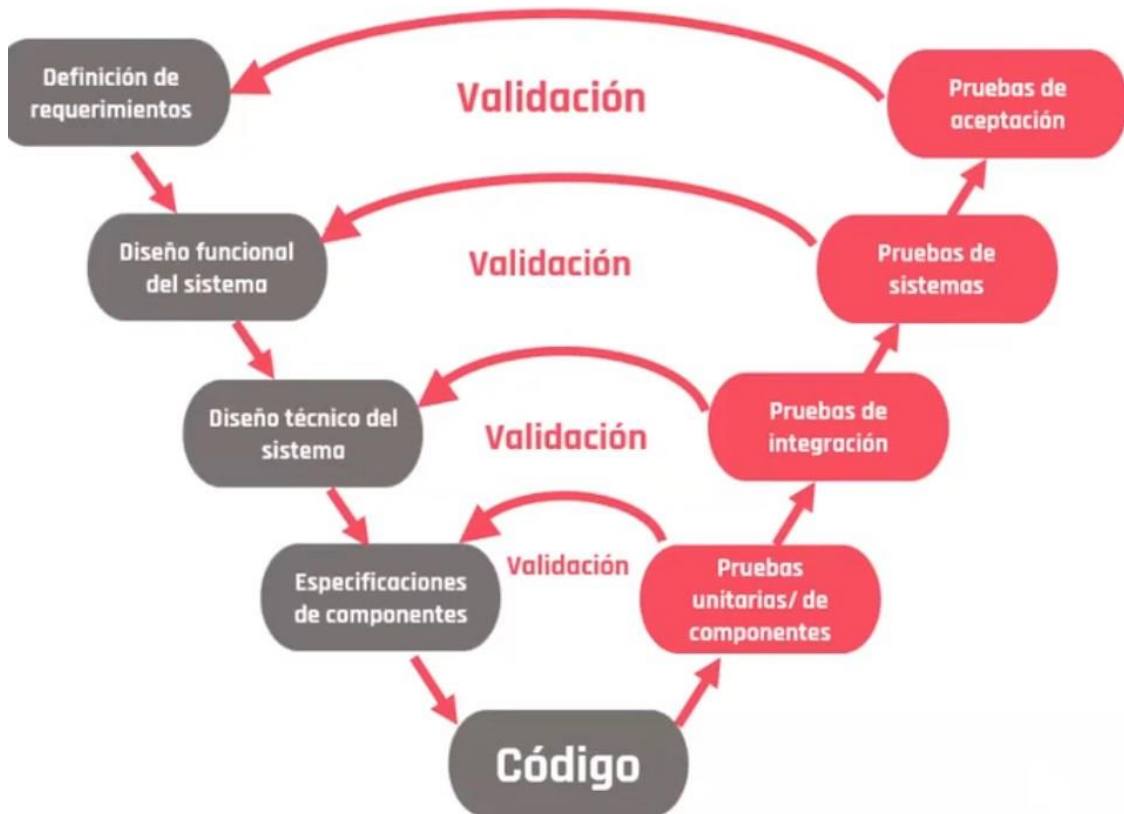
## En resumen

**Prueba de componente o unitaria:** Desde el punto de vista de desarrollo, trabajamos con una unidad que aislamos.

**Prueba de integración:** También del nivel de desarrollo, donde minimamente 2 componentes se comuniquen, estén interfaceados.

**Pruebas de sistema:** pruebas de extremo a extremo, se hace participe el area de QA.

**Prueba de aceptacion:** pruebas de extremo a extremo, con otro nivel de responsabilidad (menor). El equipo de testing participa, pero estas pruebas estan a cargo o es responsable el cliente o usuario.



## Tipos de pruebas de software:

### Pruebas funcionales

Las pruebas funcionales se llevan a cabo para comprobar las características críticas para el negocio, la funcionalidad y la usabilidad. Las pruebas funcionales garantizan que las características y funcionalidades del software se comportan según lo esperado sin ningún problema. Valida principalmente toda la aplicación con respecto a las especificaciones mencionadas en el documento Software Requirement Specification (SRS). Los tipos de pruebas funcionales incluyen pruebas unitarias, pruebas de interfaz, pruebas de regresión, además de muchas.

- Incluye pruebas que evalúan las funciones que el sistema debe realizar. Las funciones describen que hace el sistema.
- La prueba funcional observa el comportamiento del software.



## Pruebas no funcionales

Las pruebas no funcionales son como pruebas funcionales; sin embargo, la principal diferencia es que esas funciones se prueban bajo carga para el rendimiento de los observadores, fiabilidad, usabilidad, escalabilidad, etc. Las pruebas no funcionales, como las pruebas de carga y esfuerzo, normalmente se llevan a cabo mediante herramientas y soluciones de automatización, como LoadView. Además de las pruebas de rendimiento, los tipos de pruebas no funcionales incluyen pruebas de instalación, pruebas de confiabilidad y pruebas de seguridad. Estos requisitos no se exteriorizan, son tasitos. Portalibilidad, escalabilidad, performance, etc. El cliente no lo suele pedir pero se esperan que se haga.

- La prueba no funcional prueba como de bien se comporta el sistema
- El diseño y la ejecución de la prueba no funcional puede implicar competencias o conocimientos especiales, como el conocimiento de las debilidades inherentes a un diseño o tecnología.

### ¿Cual de las siguientes afirmaciones describe MEJOR las pruebas NO funcionales?

d- Las pruebas no funcionales son probar características del sistema, tales como la usabilidad, fiabilidad o mantenibilidad.

## Pruebas unitarias

Las pruebas unitarias se centran en probar piezas/unidades individuales de una aplicación de software al principio del SDLC. Cualquier función, procedimiento, método o módulo puede ser una unidad que se someta a pruebas unitarias para determinar su corrección y comportamiento esperado. Las pruebas unitarias son las primeras pruebas que los desarrolladores realizan durante la fase de desarrollo.

Como estamos en el nivel mas bajo, solucionar defectos aquí, evita que se propague a niveles superiores.

Ventajas:

- Reducen costo de pruebas
- Mejora el diseño y el código del software
- Reduce defectos en cambios recientes
- Pruebas de regresión de componentes construyen confianza en la calidad de los entregables

## Pruebas de integración

Las pruebas de integración implican probar diferentes módulos de una aplicación de software como grupo. Una aplicación de software se compone de diferentes submódulos que trabajan juntos para diferentes funcionalidades. El propósito de las pruebas de integración es validar la integración de diferentes módulos juntos e identificar los errores y problemas relacionados con ellos.

## Performance Testing

Las pruebas de rendimiento son un tipo de pruebas no funcionales, realizadas para determinar la velocidad, estabilidad y escalabilidad de una aplicación de software. Como su nombre indica, el objetivo general de esta prueba es comprobar el rendimiento de una aplicación con

respecto a los diferentes puntos de referencia del sistema y de la red, como la utilización de la CPU, la velocidad de carga de la página, el control de tráfico máximo, la utilización de recursos del servidor, etc. Dentro de las pruebas de rendimiento, hay varios otros tipos de pruebas, como pruebas de carga y pruebas de esfuerzo

## Pruebas de aceptación

En ingeniería de software y pruebas de software, las pruebas de aceptación pertenecen a las últimas etapas previas a la liberación en firme de versiones nuevas a fin de determinar si cumplen con las necesidades y/o requerimientos de las empresas y sus usuarios.

### ¿Qué es importante hacer cuando se trabaja con modelos de desarrollo de software?

a- Si fuera necesario, adaptar los modelos a las características del proyecto y del producto.

## Caja blanca, gris y negra.

Cuándo estamos refiriéndonos a “una caja” es la manera de observar el contenido del Software, ya sea que no tenemos noción más que la interfaz con la que estamos trabajando, caja negra. Cuándo podemos ver todo el contenido como una caja de cristal, caja blanca. Las integraciones, los datos cómo fluyen de un lugar a otro, dónde no conozco el código ni veo interfaz pero puedo ver cómo fluye la información a través de las redes, caja gris.

**Negra:** No podemos observar cómo fue construida, no vemos el código, no sabemos su arquitectura, no tenemos nociones más que la interfaz que estamos interactuando.

- Partición de equivalencia: en esta técnica se dividen los datos en particiones conocidas como clases de equivalencia donde cada miembro de estas clases o particiones es procesado de la misma manera. Hay clases validas y clases invalidas, las validas son las, las validas nos mostrara las particiones que el software acepte, las invalidas las que no acepte.  
Ej: RGB son los colores que acepta, cualquier otro es invalido.
- Analisis de valores límite: Es una extension de la tecnica de particion de equivalencia que solo se puede usar cuando la particion esta ordenada y consiste en datos numericos o secuenciales. Estableciendo un valor limite inferior y uno superior.
- Tabla de decisiones: Tiene que ver ocn valores de entrada que toman como salida una resultante, por ejemplo con condicion A y B se toma decisión 1. Con condicion A y C se toma decisión 2, con decisión B y C se toma decisión 3.
- Transición de estados: un diagrama de transicion de estado muestra los posibles estados del software, asi como la forma en que el software entra, sale y realiza las transiciones entre estados.  
Buscamos mostrar o excibir los diferentes estados que el software presenta. Lo importante es el ultimo estado que el software presenta.



Técnica de diseño de pruebas de caja negra en la cual los casos de prueba son diseñados para ejecutar transiciones de estado válidas e inválidas.

Estado 1	Estado 2	Estado 3	Estado 4	Estado 5	Estado final
Nacido	Soltero	Con hijo/s			Con hijo/s
Nacido	Soltero	Casado	Viudo		Viudo
nacido	soltero	casado	Divorciado	Casado	Casado

- Casos de usos

- **Participación de Equivalencia.** Esos grupos de datos que pueden entrar para casos exitosos o para casos no exitosos.

- **Valores Límite.** Se puede tener usado un rango de valores.
- **Tabla de Decisiones.** Va enfocada si tuviéramos valores seleccionables.
- **Transición de Estados.** Cómo el componente se comporta.
- **Casos de Uso.** Realizar escenarios que pueda realizar el usuario.

**Blanca:** Es como una caja de cristal, puedo ver todo lo que hay adentro e incluso puedo ser parte del equipo que desarrolla el software.

- Cobertura de declaración
- Cobertura de decisiones

- **Cobertura de declaraciones.** Las declaraciones son todo aquello que tienes dentro del código y estás asumiendo que es lo que se pide que haga, al decir cobertura es, dependiendo el tipo de software, los requerimientos, el objetivo, se establece un porcentaje de cobertura, esto significa que, cada línea de código debería ser ejecutada al menos una vez, cada sentencia debería de ejecutarse alguna vez.

- **Cobertura de código.** Que se evite tener código obsoleto.

**Gris:** Pueden ser la integraciones, cómo fluye el código y puedo ver como se transmiten los datos a través de las redes.

- Casos de negocios
- Pruebas End-to-End
- Pruebas de integración

- **Casos de Negocio.** Es necesario conocer cómo el usuario interactúa, qué datos ingresa y qué datos van a ser retornados.

- **Pruebas End to End.** Cómo se están agregando datos y aún no queremos ver datos de salida.
- **Pruebas de integración.** Ver cómo viajan esos datos, la respuesta y la comunicación de cómo fluyen los datos entre diferentes servicios.

**Pruebas basadas en la experiencia:** Aprovechan el conocimiento de desarrolladores, probadores y usuarios para diseñar, implementar y ejecutar la prueba.

Son pruebas diseñadas en base al conocimiento de los anteriormente mencionados y usan ese conocimiento para validar el flujo válido o inválido.

- Prueba de predicción de errores
- Basadas en listas de comprobación
- Pruebas de exploración o exploratorias

## Pruebas de humo y regresión.

**Están relacionadas al cambio, es decir que son dinámicas.**

**Prueba de humo** es verificar la funcionalidad core de una implementación, lo básico y que no debería estar ausente. Se realiza cuando se va a agregar un nuevo MVP, puede ser nueva funcionalidad o corrección de defectos.

**Una suite de regresión** es la verificación de la continuidad en el buen funcionamiento de otras funcionalidades, dada la implementación de otra nueva, manteniendo la integridad del sistema aunque se incluyan cambios. Se realiza cuando se va a agregar un nuevo MVP, puede ser nueva funcionalidad o corrección de defectos. Cuando automatizamos debemos empezar por las pruebas de regresión.

Dentro de las suites de regresión, van a contener lo mismo que las pruebas de humo, así que puede haber duplicidad entre ambas pruebas. Todos los test cases deberían pertenecer a una suite de humo o regresión por que ambas son importantes y se ejecutan a lo largo del desarrollo de un sistema, siendo relevantes para el funcionamiento saludable del producto en construcción y relevantes para la calidad final del producto.

## Pruebas estáticas y dinámicas:

**Estáticas:** aquellas que evaluamos de forma manual, en un tiempo específico y determinado, por eso son estáticas.

revisamos documentación, una línea de código, cosas puntuales que no necesiten la ejecución del software, pueden ser llevadas a cabo por gente que entienda del código.

Estructura, ver si un IF está cerrado, etc.

Suelen ser de caja gris, pero siempre se tiende hacia un lado o al otro en mayor medida.

### Conceptos básicos de la prueba estática

- Especificaciones, requisitos de negocio, funcionales y de seguridad.
- Epicas, historias de usuarios y criterio de aceptación. (epicas son cuestiones que hacen a la funcionalidad pero no están demasiado desglosadas, por lo cual hay que desmenuzar para atacar esos puntos uno a uno.) Ej: testear un carrito de compra y

todos sus componentes, que se cargue el carro, que se ejecute la compra, que se realice el checkout, etc.

- Especificaciones de arquitectura
- Código
- Prudctos de prueba, planes, casos procedimientos y guiones de prueba.
- Manual de usuario
- Contratos, planes de proyectos, calendarios y presupuestos.

**Dinamica:** Esta relacionado con la ejecucion del software, sistema o componente.

Puede ser basda en experiencia, se complementan con las estaticas debido a que encuentra diferentes tipos de defectos, para la generacion de casos de prueba se utilizan diferentes tecnicas de caja negra, blanca o basdadas en la experiencia de usuario.

**Proceso de revision:** seleccionamos un producto de trabajo cuidadosamente, su trabajo es encontrar y remover errores, puede ser realizadas por uno o mas personas.

Revisiones formales: roles definidos, siguen un proceso establecido y deben ser documentadas. Ej: auditorias programadas.

Una revision es de las mas formales. CREO

Revisiones informales: no siguen un proceso definido y no son documentadas formalmente. Ej: empresas pequeñas donde todos hacen todo y dependiendo de demandas/disponibilidad/requisitos se tapan baches con estas revisiones informales. No hay plazos claros y documentacion escasa o nula.

Requisitos /requerimientos: definen condiciones de funcionamiento del sistema en el ambiente operacional.

- Requisito de usabilidad
- Eficiencia
- Disponibilidad
- Confiabilidad
- Integridad
- Mantenibilidad

Lo evalúa un test manager

## Tipos de prueba según ambiente



**No desarrollaremos en ambiente de producción debido a que:**

EN general los probadores no tienen acceso a este ambiente

en el caso de tener acceso y realizar pruebas

no se deben realizar acciones que generen datos

Se corre el riesgo de ingresar datos basura

Se interfiere en los datos de seguimiento.

### Debugging

Es depurar un código, encontramos, analizamos y removemos las causas de fallos de software.

### Breakpoint

Un breakpoint es una pausa en nuestro código para entender en qué estado están nuestras variables. Donde frenamos el debug para analizar y sacar conclusiones.

Un breakpoint tiene un punto rojo para indicar donde establecemos un breakpoint, tiene un número de línea que indica la posición en el código, y el contenido que es la línea de código.

### Diferencias entre testing y debugging

El testing lo hace QA, el debugging lo hace el desarrollador.

El testing se puede automatizar, el debugging es manual.

El testing encontramos y reportamos los defectos, en el debugging encontramos el problema y lo arreglamos.

### Pruebas de componentes

#### Técnicas de prueba de caja blanca

También conocidas como pruebas estructurales, se basan en la estructura interna del objeto de prueba, es decir, que están fuertemente ligados al código fuente.

Esta tecnica se puede utilizar en todos los niveles de prueba.

Cuando se crea casos de prueba con este tipo de tecnicas es aconsejable utilizar tambien las tecnicas de caja negra como particion de equivalencia y analisis de valores limites. De este modo se conseguiran datos de prueba que maximicen la cobertura de prueba.

**Las siguientes pruebas se realizan con mayor frecuencia en el nivel de prueba de componenete.**

#### **Prueba y cobertura de sentencia.**

Cuando hablamos de cobertura de sentencia, nos referimos al porcentaje de sentencias ejecutables que han sido practicadas por un juego de pruebas. Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute al menos una vez.

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

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

#### **Prueba y cobertura de decisión**

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

- Ejercita las decisiones en el codigo y prueba el codigo que se ejecuta basado en los resultados de la decision.
- Los casos de prueba siguen los flujos de control que se producen desde un punto de decision.
- En el caso de un IF se necesitan dos casos de prueba como minimo, uno para el valor VERDADERO y otro para el FALSO de la decision.
- En el caso de un case se necesitan casos de prueba para todos los resultados posibles, incluido el por defecto.
- Ayudan a encontrar defectos en el codigo que no fueron practicados por otras pruebas ya que se debenr ecorrer todos los caminos de una decision.

- Cuando se alcanza el 100% de cobertura de decision, se ejecutan todos los resultados de decision. Esto incluye probar el resultado verdadero y tambien el resultado falso, incluso cuando no hay una sentencia falsa explicita.
- Lograr una cobertura del 100% de decision garantiza una cobertura del 100% de sentencia, pero no al revés.
- La cobertura se mide como:

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

## TDD

El desarrollo agil exige feedback periodico. Se puede denominar como “desarrollo impulsado por retroalimentacion”. Consiste en trabajar conjuntamente con el cliente y las areas usuarias para interpretar y comunicar correctamente lo que se espera desde el punto de vista del negocio.

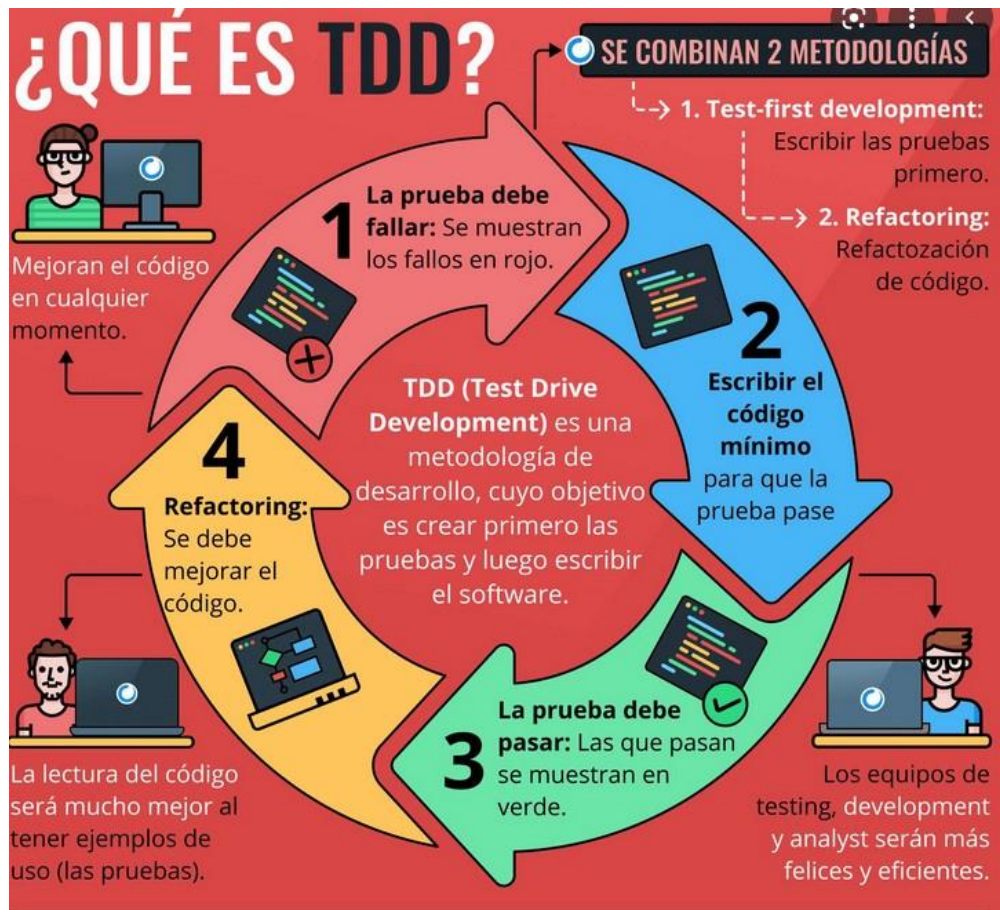
Puede que los requisitos del proyecto cambien en cualquier momento. Para lidiar con productos alineados con los requisitos cambiantes del cliente, necesitamos retroalimentacion constante para evitar distribuir software inutilizable.

TDD (test driven development o desarrollo guiado por pruebas) cobra importancia, dado que esta diseñado para ofrecer tal retroalimentacion desde el momento inicial del proyecto.

TDD consiste en escribir primero las pruebas unitarias, luego el codigo fuente que pasa la prueba satisfactoriamente y, por ultimo, refactorizar el codigo escrito. Asi, su logica siguen el camino inverso al desarrollo tradicional, en el que habitualmente se codifica y luego se verifica el software.

Este enfoque que TDD propone, mitiga los cuellos de botella criticos que obstruyen la calidad y la entrega del software. Tiene base en la retroalimentacion, correccion de errores y adiccion de nuevas funciones, el sistema evoluciona para garantizar que todo funcione según lo esperado. TDD mejora la colaboracion dentro de los equipos, tanto del desarrollo como de control de calidad, asi como con el cliente. Tambien se ahorra tiempo, ya que los equipos no necesitan insumir tiempo adicional recreando extensos scripts y suites de prueba.

- El metodo tradicional planea tomar funciones y componentes, analizar sus casos de uso y escribir test cubriendo las distintas alternativas encontradas.
- TDD propone que lo primero que se debe hacer es escribir los tests y luego codificar el software.



TDD propone pensar y comprender primero el problema en su totalidad, antes de plantear la solución.

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





## Desarrollo guiado por pruebas (TDD)

El proceso consiste en:

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

## Desarrollo guiado por el comportamiento (BDD)



El proceso consiste en:

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

Desarrollo guiado por pruebas (TDD)	Desarrollo guiado por el comportamiento (BDD)
Es un proceso de desarrollo de software donde se desarrolla el código guiado por casos de prueba automatizados.	Es un proceso de desarrollo de software que permite al desarrollador concentrarse en probar el código basándose en el comportamiento esperado del software.
Las pruebas escritas son principalmente de nivel unitario y se centran en el código, aunque también pueden escribirse pruebas a nivel de integración o de sistema.	Las pruebas escritas son principalmente de nivel de sistema e integración, aunque también se pueden utilizar para escribir pruebas unitarias.
Ayuda a los desarrolladores a concentrarse en resultados esperados claramente definidos.	Ayuda al desarrollador o probador a colaborar con otras partes interesadas para definir pruebas precisas centradas en las necesidades del negocio.
Existe menor redundancia debido a que las pruebas se automatizan y se utilizan en la integración continua.	Existe menor retrabajo debido a que las pruebas suelen ser más fáciles de entender para los demás miembros del equipo y los implicados.
Mayor calidad en el código desarrollado.	Mayor calidad en software debido a que todo el equipo puede entender y colaborar en las pruebas.



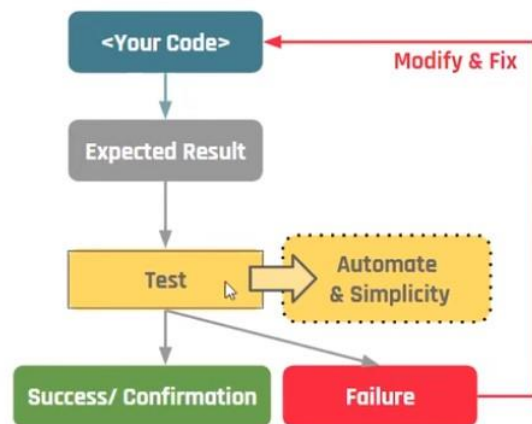
Productividad productividad debido a que hay un menor tiempo de debugging. Menor comunicación debido a que para entender las pruebas se necesita conocer un lenguaje técnico.	Mayor productividad debido a que los casos de prueba se pueden compartir con todas las partes interesadas y los frameworks utilizados generan métricas en forma automática. Mejora la confianza entre los miembros del equipo. Mayor retroalimentación con el cliente.
---	--

## Proceso de BDD

### Proceso

En general, 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.

## Proceso TDD

En el proceso tdd, los test estan en la primera linea, no el codigo.

### Automatización

- Test Runner
- Assertion Library

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

### Frameworks

- JUnit
- Nunit
- JMockit



- EMMA
- PHPUnit



## BackEnd testing

Nos garantiza que los datos contenidos en la base de datos de una aplicación y su estructura satisfagan los requisitos del proyecto.

## Api testing

Estas pruebas consisten en hacer peticiones HTTP (GET, POST, PUT, DELETE) y luego verificar la respuesta.

Se utiliza POSTMAN

## Debug:

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

Se invierte mucho tiempo en encontrar y depurar estos errores. Pueden ser leves o graves.

## Tipos:

- **Deporacion por fuerza bruta:** se invocan señales en tiempo de ejecución y se carga el programa con instrucciones de salida, entonces en algún lugar del código donde se produce el error se espera encontrar una pista que pueda conducir a la causa del error.
- Esto significa a fines práctico ir dejando pequeños mensajes como un `console.log` a lo largo del código y vamos comparando esos log con lo que esperaríamos en la salida, de esta manera podríamos hasta hacer un análisis un poco más detallado de lo que contienen nuestras variables para entender que está sucediendo.
- **Backtracking:** Es un enfoque de depuración muy común usado en pequeños programas, empezando en el sitio donde se descubrió un síntoma y se recorre manualmente hacia atrás el código fuente hasta llegar a la causa. Si aumenta el número de líneas, el camino hacia atrás se hace tan grande que es inmanejable.
- **Eliminación de causas:** los datos relacionados con el error se usan para aislar la causa posible, tratando de elaborar una hipótesis de las causas. Se utilizan los datos mencionados para validar dicha hipótesis.



Todas estas herramientas se complementan con las herramientas de depuración.







Los IDE proporcionan la manera de capturar errores típicos específicos del lenguaje como caracteres de final de línea, variables indefinidas, entre otros problemas, sin requerir compilación. Cuentan con compiladores con depuración, ayudas dinámicas para la depuración, Generadores automáticos de casos de pruebas y Herramientas de correlación de referencias cruzadas.

Sin embargo, esto no es reemplazo de una evaluación cuidadosa basada en un modelo del diseño y un código fuente claro.

## Breakpoint

Es un punto de interrupción en nuestro código para detener la ejecución del programa en líneas específicas ya analizar la situación del mismo, revisando por ejemplo el estado de las variables o de la pila de llamadas en ese momento.

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

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.

### Prueba de componente

La prueba de componente o prueba unitaria es la prueba de los componentes individuales de software. 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 automatizadas escritas y ejecutadas por desarrolladores de software para garantizar una sección de una aplicación, conocida como la “unidad”, cumpla con su diseño y se comporte según lo previsto.

El proceso de 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).
- 4.

### **¿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.

### **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.

Aunque **TDD** o **BDD** parecen muy similares, la principal diferencia entre ambas está en el alcance. **TDD** es una práctica de desarrollo, enfocada en cómo escribir el código y cómo debería funcionar. Mientras que **BDD** es un enfoque de equipo que hace hincapié en por qué debes escribir ese código y cómo se debería comportar

### **Generalidades**

La prueba de componenete, 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, virtualizacion de servicios, arneses, stubs y controladores.

Este tipo de pruebas puede cubrir:

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

### **Hablando de cobertura**

Hablamos de calidad en el código de software, a eso nos referimos, a nivel de caja blanca si estamos trabajando con Jest, tenemos dos tipos de coberturas, las cuales se basan en el flujo de control.

**Cobertura de sentencia:** cantidad de líneas que está leyendo mi test, que pasaran correctamente y que a su vez generen un resultado de calidad. Cobertura es un cociente, es un porcentaje que queremos tienda a ser un 100%, pero es imposible llegar a 100% fácilmente, el porcentaje idóneo para tener una buena cobertura depende del proyecto, pero se estiman valores entre 70-80%.

Centrado en la cobertura de sentencias del código: ¿Qué casos de prueba son necesarios con el objeto de ejecutar todas o un % de las sentencias de un código existente?

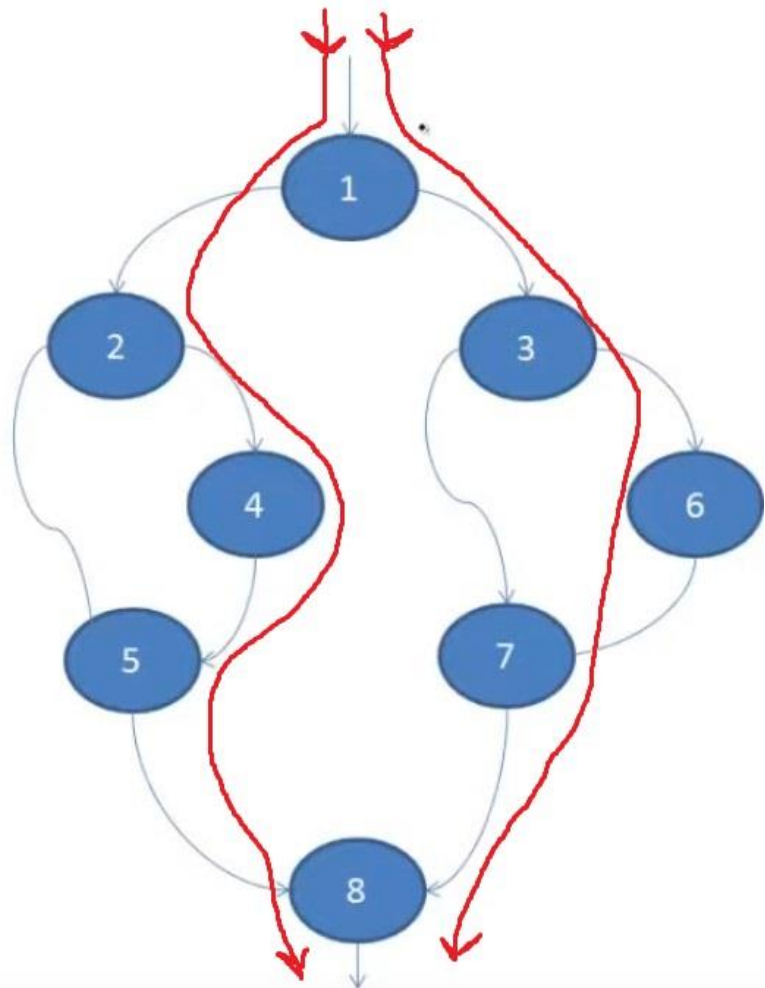
Las sentencias están representadas por nodos, el flujo de control está representado por flechas.

La cobertura de sentencias  $\rightarrow C_s$

$$C_s = \frac{\text{Número de sentencias visitadas}}{\text{Número total de sentencias}} * 100\%$$

Cuántos casos de prueba se requieren para lograr una cobertura de sentencia del 100%?

Se necesitan dos casos de prueba para alcanzar el 100% de la cobertura de sentencia.





El objetivo es escribir el menor numero de casos de pruebas para cubrir la totalidad de los nodos. Es decir visitar todos los nodos al menos una vez.

**Cobertura de decisión:** Aquella cobertura que me indica que recorrimos todas las lineas dentro de una estructura de if, por ejemplo camino verdadero, camino falso. De esta manera cubrimos todos los caminos posibles dentro de esa estructura de IF.

Si tenemos un 100% de cobertura de decisión, nuestro porcentaje de cobertura de sentencia daria el 100%.

A nivel de decisión testeamos el 100% de mis lineas, garantizando un 100% de sentencia. Lo cual al revés no es estrictamente igual, podriamos tener cobertura de sentencia del 100% pero no 100% de decisión.

En vez de concentrarse en los nodos o las sentencias, se centra en todas las aristas del diagrama de flujo de control y deben ser visitadas por lo menos una vez. Esta basada en el flujo de control.

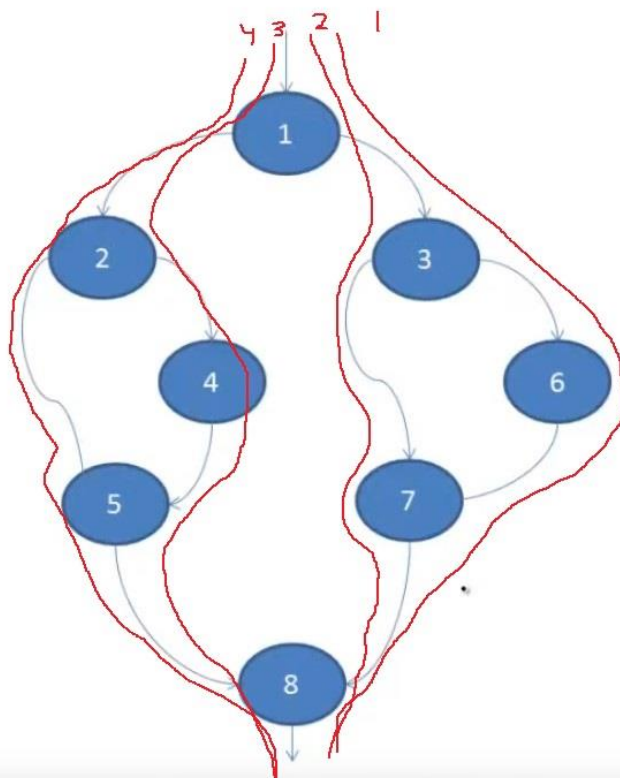
- Cobertura de Decisión/Rama → Cr

$$Cr = \frac{\text{Número de ramas visitadas}}{\text{Número total de ramas}} * 100\%$$

El objetivo de esta técnica (criterio de salida) es lograr la cobertura de un % específico de todas las decisiones

Cuántos casos de prueba se requieren para lograr una cobertura de decisión del 100%?

Se necesitan 4 pruebas para una cobertura del 100%, casi siempre es mas exhaustiva que la cobertura de sentencia.



## Primer kahoot

1. Cuales son los niveles de prueba?  
Componente-integracion-sistema-aceptacion.
2. Uno de los objetivos de la prueba de aceptacion es establecer confianza en la calidad del sistema en su conjunto.  
Verdadero.
3. La prueba no funcional observa el comportamiento del software.  
Falso
4. Un caso de uso es la base de prueba para las pruebas de sistema.  
Verdadero.
5. Una de las bases de prueba para las pruebas de componente es el codigo.  
Verdadero
6. La prueba funcional prueba "que tan bien" se comporta el sistema.

## Segundo kahoot

1. ¿Que es una tecnica de prueba?  
Una guia, Como vamos a trabajar las pruebas, Un modelo para entregar software de calidad.
2. Particion de equivalencia es una tecnica de...  
Caja Negra.
3. Analisis de valores limites es una tecnica de caja blanca  
Falso.
4. Pruebas basadas en comprobacion es una tecnica de Caja Blanca  
Falso.
5. ¿A que clasificacion responden las pruebas basadas en listas de comprobacion.  
Basadas en la experiencia.

### Ejercicio PG

¿Por qué es necesario aplicar tecnicas de pruebas para el diseño de software?

Nos enfocamos en como vamos a trabajar las pruebas de software. Posteriormente revisarlas, analizarlas, diseñarlas, implementarlas y ejecutarloas para nuestro software bajo alcance.

¿Cuál es la diferencia entre pruebas de caja negra y de caja blanca?

Las tecnicas de caja negra no tarbajan con el codigo, pero si desde la interfaz de usuario, mientras que las tecnicas de caja blanca si trabajan con la revision del codigo y su ejecucion bajo diferentes instancias.



¿Qué relación existe entre la partición de equivalencia y el análisis de valores límites?  
Ambas definen las clases válidas e inválidas, ambas son de caja negra.  
Mientras la partición de equivalencia acepta cualquier valor de datos, mientras que los valores límites aceptan valores numéricos detallados.

Explique brevemente el significado de una clase de equivalencia no válida.  
Son aquellas clases que el software estará rechazando, es decir, no aceptando como tales.

¿Por qué es necesario aplicar técnicas de prueba por el diseño de las mismas?

Nos enfocamos en cómo vamos a trabajar las pruebas de software, posteriormente revisarlas, analizarlas, diseñarlas, implementarlas y ejecutarlas para nuestro software bajo alcance.

## **Automatización de la prueba**

¿Por qué vamos a automatizar una prueba?

Ganamos tiempo durante la ejecución de la prueba

Eliminar o minimizar el error humano

Reducción en el costo de ejecución a largo plazo

Hay pruebas que se pueden ejecutar automáticamente, pero no manualmente. Donde no se pueda más manualmente, lo automatizamos.

¿Qué podemos automatizar?

Pruebas de API, pruebas unitarias, pruebas de performance pasando por pruebas de UI

Se pueden automatizar muchas cosas grabando macros, invirtiendo tiempo al inicio.

El ojo del ser humano siempre tiene que estar dando vueltas.

Enfocamos nuestra energía en pruebas de interfaz gráfica.

Cuando pensamos en automatizar no hablamos de grandes scripts, hablamos de automatizar e ir testeando pequeñas partes.

Ventajas

pruebas complejas

Ahorra tiempo

Las pruebas están menos sujetas a errores del tester.

## Desventajas

Necesidad de conocimiento tecnico para realizarse

Mantenimiento continuo

Existen criterios de calidad que no pueden ser automatizados.