# **Introducción al Testing de Software**

Resumen

En este apunte de clase se introducen brevemente los conceptos básicos de testing de software necesarios para comprender las técnicas de testing que se enseñan más adelante.

1. Verificación y validación

El testing de software pertenece a una actividad o etapa del proceso de producción de software denominada Verificación y Validación –usualmente abreviada como V&V.

V&V es el nombre genérico dado a las actividades de comprobación que aseguran que el software respeta su especificación y satisface las necesidades de sus usuarios. El sistema debe ser verificado y validado en cada etapa del proceso de desarrollo utilizando los documentos (descripciones) producidas durante las etapas anteriores.

En rigor no solo el código debe ser sometido a actividades de V&V sino también todos los subproductos generados durante el desarrollo del software.

Estos subproductos deben ser verificados de forma tal de que exista mayor confianza en que cumplen con los requerimientos del cliente, en particular el equipo de desarrollo debe asegurarse de que la arquitectura podrá incorporar los cambios previstos a bajo costo y que esta habilita otras propiedades que haya solicitado el cliente tales como seguridad, portabilidad, etc. Tareas semejantes deben llevarse a cabo para los otros subproductos del desarrollo. Si bien estos términos en su uso cotidiano pueden llegar a ser sinónimos, en Ingeniería de Software tienen significados diferentes y cada uno tiene una definición más o menos precisa.

**Validación**: ¿estamos construyendo el producto correcto?

**Verificación**: ¿estamos construyendo el producto correctamente?

En este sentido, la verificación consiste en corroborar que el programa respeta su especificación, mientras que validación significa corroborar que el programa satisface las expectativas del usuario. En otras palabras, la verificación es una actividad desarrollada por ingenieros teniendo en cuenta un modelo del programa y el programa en sí, en tanto que la validación la debe realizar el usuario teniendo en cuenta lo que él espera del programa y el programa en sí.

Existen varias técnicas dentro del marco de la V&V desde las más informales (prototipación de requerimientos, revisión de requerimientos, etc.), pasando por las semiformales (el testing es la más conocida pero ciertas técnicas de análisis estático de código se usan frecuentemente), hasta la prueba formal de programas, el cálculo de refinamiento, etc. Aunque el testing se puede aplicar tanto en verificación como en validación, en este curso nos concentraremos casi exclusivamente en el testing de programas como parte de la verificación. Es decir que no estudiaremos otras técnicas de verificación, no estudiaremos técnicas de validación y no estudiaremos la V&V de otras descripciones más allá del programa.

1. Definición de testing y vocabulario básico

El testing es una actividad desarrollada para evaluar la calidad del producto y para mejorarlo al identificar defectos y problemas. El testing de software consiste en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados a partir del dominio de ejecución que usualmente es infinito, en relación con el comportamiento esperado. Las palabras resaltadas en el párrafo anterior corresponden a las características fundamentales del testing. Es una técnica dinámica en el sentido de que el programa se verifica poniéndolo en ejecución de la forma más parecida posible a como ejecutará cuando esté en producción esto se contrapone a las técnicas estáticas las cuales se basan en analizar el código fuente. El programa se prueba ejecutando solo unos pocos casos de prueba dado que por lo general es física, económica o técnicamente imposible ejecutarlo para todos los valores de entrada posibles.

**“Si uno de los casos de prueba detecta un error(1), el programa es incorrecto, pero si ninguno de los casos de prueba seleccionados encuentra un error no podemos decir que el programa es correcto (perfecto)”**.

Esos casos de prueba son elegidos siguiendo alguna regla o criterio de selección. Se determina si un caso de prueba ha detectado un error o no comparando la salida producida con la salida esperada para la entrada correspondiente, la salida esperada debería estar documentada en la especificación del programa. Las limitaciones antes mencionadas no impiden que el testing se base en técnicas consistentes, sistemáticas y rigurosas. Sin embargo, en la práctica industrial, como ocurre con otras áreas de Ingeniería de Software, usualmente se considera solo una parte mínima de dichas técnicas tornando a una actividad razonablemente eficaz y eficiente en algo fútil y de escaso impacto. Aunque en la industria de software el testing es la técnica predominante en aquellos casos minoritarios en que se realiza una actividad seria de V&V, en el ambiente académico se estudian muchas otras técnicas.

Veamos algunos conceptos básicos de testing. Testear un programa significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados. El testing se estructura en casos de prueba o casos de test; los casos de prueba se reúnen en conjuntos de prueba. Desde el punto de vista del testing se ve a un programa (o subrutina) como una funci´on que va del producto cartesiano de sus entradas en el producto cartesiano de sus salidas. Es decir:

P : ID → OD

donde ID se llama dominio de entrada del programa y OD es el dominio de salida. Normalmente los dominios de entrada y salida son conjuntos de tuplas tipadas cuyos campos identifican a cada una de las variables de entrada o salida del programa, es decir:

ID = [ b x1 : X1, . . . , xn : Xn]

OD = [ b y1 : Y1, . . . , ym : Ym]

De esta forma, un caso de prueba es un elemento, x , del dominio de entrada (es decir x ∈ ID) y testear P con x es simplemente calcular P(x ). En el mismo sentido, un conjunto de prueba, por ejemplo T, es un conjunto de casos de prueba **definido por extensión** y testear P con T es calcular P(x ) para cada x ∈ T. Es muy importante notar que x es un valor constante, no una variable; es decir, si por ejemplo ID = [ b num : N] entonces un caso de prueba es 5, otro puede ser 1000, etc., pero no n > 24.

También es importante remarcar que x1, . . . , xn son las entradas con que se programó el programa (esto incluye archivos, parámetros recibidos, datos leídos desde el entorno, etc.) y no entradas abstractas o generales que no están representadas explícitamente en el código. De la misma forma, y1, . . . , ym son las salidas explicitas o implícitas del programa (esto incluye salidas por cualquier dispositivo, parámetro o valor de retorno, e incluso errores tales como no terminación, Segmentación fault, etc.).

De acuerdo a la definición clásica de corrección, un programa es correcto si verifica su especificación**(2)**. Entonces, al considerarse como técnica de verificación el testing, un programa es correcto si ninguno de los casos de prueba seleccionados detecta un error. Precisamente, la presencia de un error o defecto se demuestra cuando P(x ) no satisface la especificación para algún x en ID. Una falla es el síntoma manifiesto de la presencia de un error. Es decir que un error**(3)** permanecerá oculto hasta que ocurra una falla causada por aquel. Por ejemplo, si la condición de una sentencia if

1. En el área de testing se distingue con cierta precisión los términos error o defecto, falla y falta, aunque por el momento nosotros los consideraremos sinónimos.
2. Esta definición se formalizará hasta cierto punto en secciones posteriores.
3. Falla es nuestra traducción para failure.

es x > 0 cuando debería haber sido x > 1, entonces hay un error en el programa, que se manifestará (falla) cuando se testee el programa con x igual a 1 si por fortuna este caso de prueba es seleccionado porque en tal caso aparecerá cierto mensaje en la pantalla que para ese valor de x no debería haber aparecido. En este sentido el testing trata de incrementar la probabilidad de que los errores en un programa causen fallas, al seleccionar casos de prueba apropiados. Finalmente, una falta es un estado intermedio incorrecto al cual se llega durante la ejecución de un programa, por ejemplo, se invoca a una subrutina cuando se debería haber invocado a otra, o se asigna un valor a una variable cuando se debería haber asignado otro**(4)**.

3. El proceso de testing

Es casi imposible, excepto para los programas más pequeños, testear un software como si fuera una única entidad monolítica. Los grandes sistemas de software están compuestos de subsistemas, módulos y subrutinas. Se sugiere, por lo tanto, que el proceso de testing esté guiado por dicha estructura, como muestra la Figura 1. Más aún, si el proceso sugerido se combina adecuadamente con el proceso de desarrollo, como muestra la Figura 3, entonces se habilita la posibilidad de ir detectando errores de implementación lo más tempranamente posible lo que a su vez reduce el costo de desarrollo.

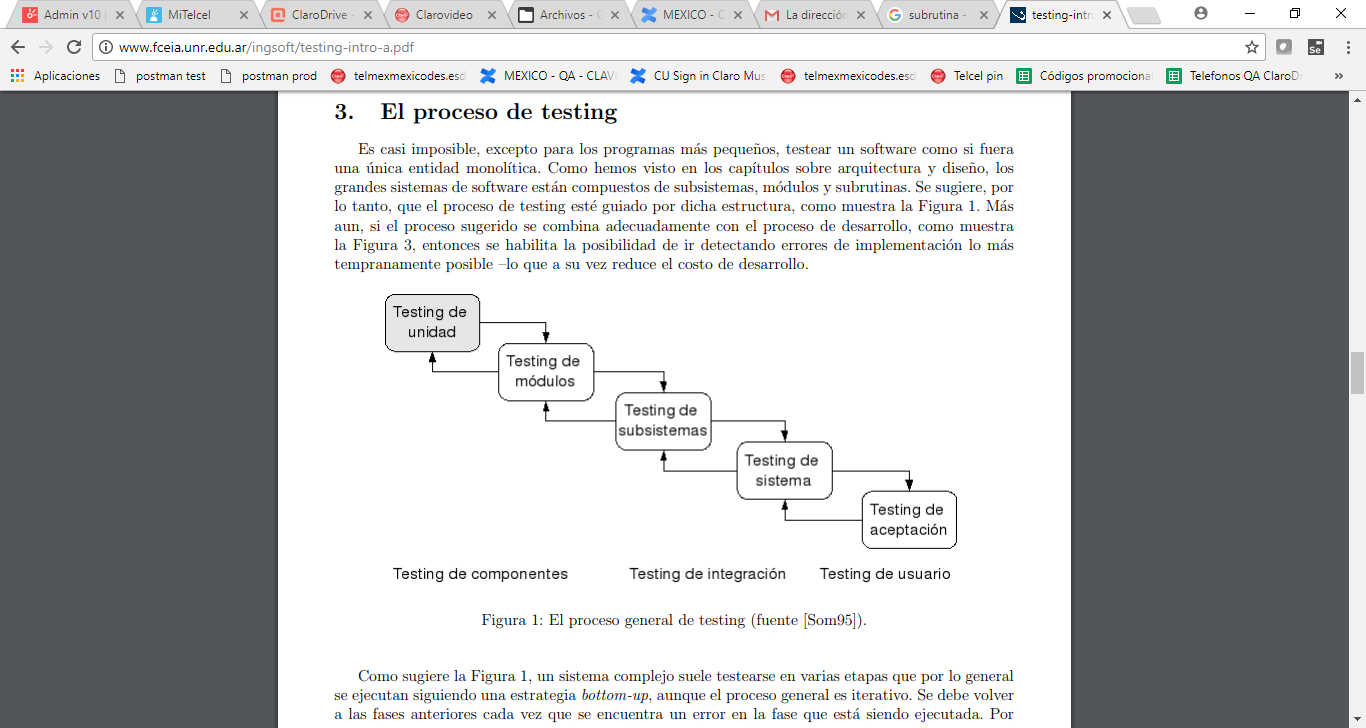
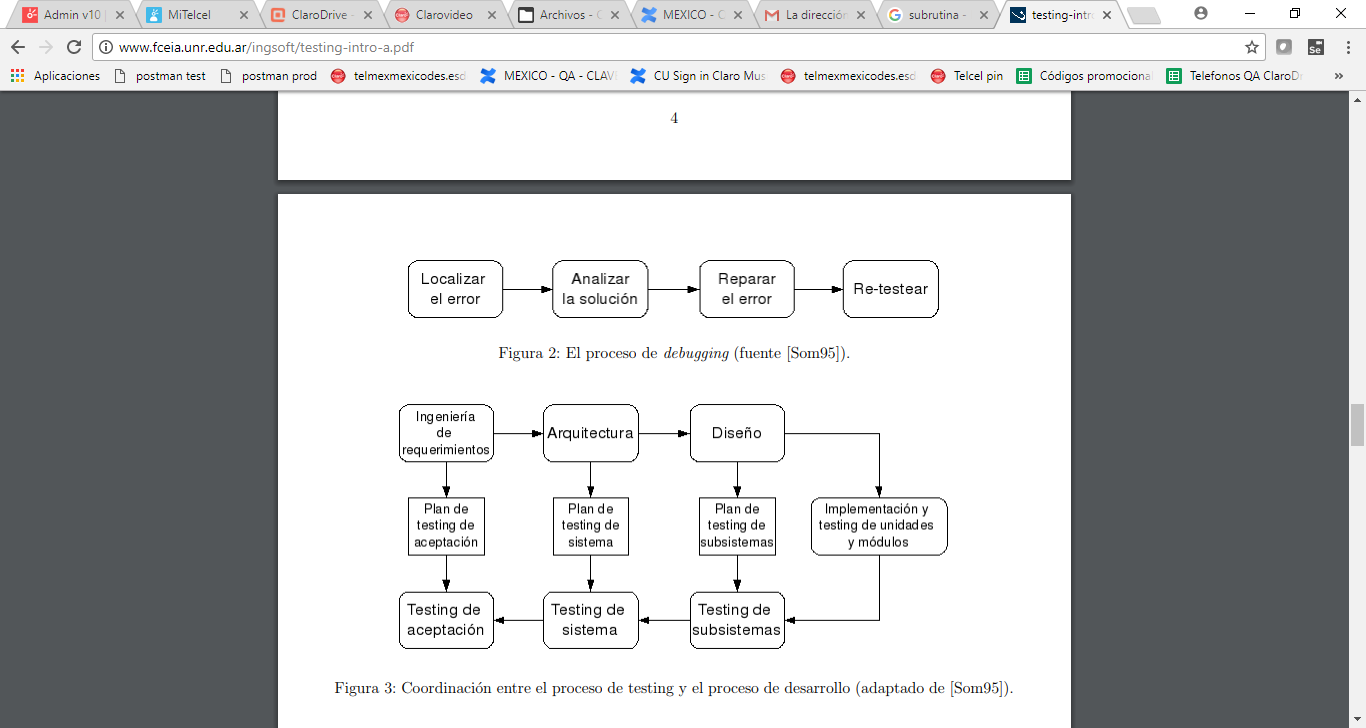


Figura 1: El proceso general de testing. Como sugiere la Figura 1, un sistema complejo suele testearse en varias etapas que por lo general se ejecutan siguiendo una estrategia bottom-up, aunque el proceso general es iterativo. Se debe volver a las fases anteriores cada vez que se encuentra un error en la fase que está siendo ejecutada. Por ejemplo, si se encuentra un error durante el testing de un subsistema particular, una vez que aquel haya sido reparado se deberán testear la unidad donde estaba el error y luego los módulos al cual pertenece la unidad reparada. En general, una vez detectado un error se sigue el proceso graficado en la Figura 2. De aquí que a las iteraciones del proceso de la Figura 1 se las llame re-testing. Dado que no hay una definición precisa de subsistema e incluso de unidad, el proceso de testing sugerido debe ser considerado como una guía que debe ser adaptada a cada caso específico. El testing de aceptación mencionado en la figura pertenece a la validación de un sistema (y no a la verificación)



\*\*\*\*\*Falta es nuestra traducción para fault.\*\*\*\*\*

dado que es el usuario el que usa el sistema en un entorno más o menos real con el fin de comprobar si es una implementación razonable de los requerimientos.

En este curso nos concentraremos en el testing de unidad dado que es la base para el resto del proceso.

En la práctica industrial usualmente se entiende que el testing es una actividad que se realiza una vez que los programadores han terminado de codificar; en general es sinónimo de testing de aceptación. Entendido de esta forma el testing se convierte en una actividad costosa e ineficiente desde varios puntos de vista:

* Los testers estarán ociosos durante la mayor parte del proyecto y estarán sobrecargados de trabajo cuando este esté por finalizar.
* Los errores tienden a ser detectados muy tarde.
* Se descubre un gran número de errores cuando el presupuesto se está terminando.
* Los errores tienden a ser detectados por los usuarios y no por el personal de desarrollo lo que implica un desprestigio para el grupo de desarrollo.

Por lo tanto, se sugiere un proceso de testing mejor imbricado con el proceso general de desarrollo, como se muestra en la Figura 3. En cuanto el equipo de desarrollo cuenta con un documento de requerimientos más o menos estable, los testers pueden comenzar a definir casos de prueba para el testing de aceptación, dado que este se basa en validar los requerimientos. De la misma forma, tan pronto como se ha definido la arquitectura o el diseño del sistema, los testers pueden usar dicha documentación para calcular casos de prueba para el testing de subsistemas y del sistema. Incluso teniendo un diseño más o menos detallado descripto con la documentación que hemos analizado en capítulos anteriores y una especificación (formal o informal) de cada unidad, es perfectamente 5 posible calcular casos de prueba de unidad mientras los programadores hacen su trabajo. De esta forma se reduce notablemente el impacto de algunos de los problemas enumerados más arriba.

Aunque la literatura de Ingeniería de Software propone una y otra vez este tipo de procesos, en nuestra opinión las técnicas y herramientas concretas disponibles actualmente aun están muy lejos como para hacer del testing una actividad rentable para la mayoría de los equipos de desarrollo.

Más aun, estamos convencidos de que la llave para habilitar estos procesos radica en contar con la posibilidad de automatizar el testing de unidad de alguna forma tal que los Tester puedan comenzar a calcular los casos de prueba desde las fases iniciales del proyecto.

1. Testing de distintos aspectos de un software

Hasta aquí hemos tratado de manera implícita qué es lo que se verifica en un software. Por otro lado, hemos definido que un programa es correcto si verifica su especificación. Pero, ¿qué describe la especificación de un software?

Al hablar de especificación hacemos referencia únicamente a la función del software, es decir a las operaciones que el sistema debe realizar, el orden en que debe realizarlas, etc. Sin embargo, la especificación de un software, en términos más generales, incluye o debería incluir otros aspectos (o requerimientos no funcionales) más allá de la funcionalidad. Por ejemplo, suelen especificarse requisitos de seguridad, desempeño, tolerancia a fallas, usabilidad, etc. Por lo tanto, al decir que un programa es correcto si verifica su especificación en realidad estamos incluyendo todos esos aspectos además de la funcionalidad. En consecuencia, al testear un programa deberíamos seleccionar casos de prueba con el objetivo de testear todos esos aspectos y no solo la funcionalidad.

Es decir, no solo deberíamos intentar responder preguntas tales como “¿El sistema permite cargar una transacción solo en las condiciones especificadas?”, sino también otras como “¿El sistema permite cargar hasta 10.000 transacciones por minuto?”, “¿El sistema permite que solo los usuarios autorizados carguen transacciones?”, etc. Claramente todo esto es posible si el equipo de desarrollo cuenta con las especificaciones correspondientes. El testing de algunos de estos aspectos no funcionales se ha especializado en alguna medida dando lugar a áreas más específicas con sus propias metodología, técnicas y herramientas. El caso más reconocido es, tal vez, el testing de seguridad el cual intenta encontrar errores que permitan a usuarios no autorizados utilizar o modificar datos o programas. El testing de este aspecto se ha especializado tanto que los testers de seguridad son profesionales con una preparación muy específica para realizar esta tarea. A pesar de todas estas consideraciones en este capítulo solo abordaremos el testing de la funcionalidad de un software.

1. Las dos metodologías clásicas de testing

Tradicionalmente el testing de software se ha dividido en dos estrategias básicas que se supone son de aplicación universal.

Testing estructural o de caja blanca. Testear un software siguiendo esta estrategia implica que se tiene en cuenta la estructura del código fuente del programa para seleccionar casos de prueba es decir, el testing está guiado fundamentalmente por la existencia de sentencias tipo if, case, while, etc.

En muchas ocasiones se pone tanto ´énfasis en la estructura del código que se ignora la especificación del programa, convirtiendo al testing en una tarea un tanto desprolija e inconsistente.

Como los casos de prueba se calculan de acuerdo a la estructura del código, no es posible generarlos sino hasta que el programa ha sido terminado. Peor aún, si debido a errores o cambios en las estructuras de datos o algoritmos aun sin que haya cambiado la especificación, puede ser necesario volver a calcular todos los casos. Por estas razones preferimos la otra estrategia de testing, aunque estudiaremos el testing estructural con cierto detalle. Sin embargo, es interesante remarcar la racionalidad detrás de esta estrategia: no se puede encontrar un error si no se ejecuta la línea de código donde se encuentra ese error –aunque ejecutar una línea de código con algunos casos de prueba no garantiza encontrar un posible error; piense en una asignación de la forma x = 1/y. Por consiguiente, es importante seleccionar casos de prueba que ejecuten al menos una vez todas las líneas del programa, lo cual se logra analizando la estructura del código. Se dice que el testing estructural prueba lo que el programa hace y no lo que se supone que debe hacer.

Testing basado en modelos o de caja negra**(5)** . Testear una pieza de software como una caja negra significa ejecutar el software sin considerar ningún detalle sobre cómo fue implementado. Esta estrategia se basa en seleccionar los casos de prueba analizando la especificación o modelo del programa, en lugar de su implementación. Algunos autores consideran que el testing basado en modelos (MBT) es la automatización del testing de caja negra. Para lograr esta automatización el MBT requiere que los modelos sean formales dado que esta es la única forma que permite realizar múltiples análisis mecánicos sobre el texto de los modelos. Por el contrario, el testing de caja negra tradicional calcula los casos de prueba partiendo del documento de requerimientos. Como en el MBT los casos de prueba se calculan partiendo del modelo, es posible comenzar a “testear” casi desde el comienzo del proyecto al menos mucho antes de que se haya terminado de programar la primera unidad. Por otra parte, pero por la misma razón, los casos de prueba calculados con técnicas de MBT son mucho más resistentes a los cambios en la implementación que aquellos calculados con técnicas de testing estructural. Más aun, el MBT es, efectivamente, automatizable en gran medida como demostraremos más adelante al utilizar una herramienta para tal fin. La gran desventaja del MBT radica en su misma definición: se requiere de un modelo formal del software, cosa que solo un porcentaje ínfimo de la industria es capaz de realizar. Debido al énfasis que le hemos dado a los métodos formales y debido a las ventajas que presenta el MBT es que estudiaremos con mucho detalle una técnica específica de esta estrategia de testing. Se dice que el testing basado en modelos prueba lo que el programa se supone que debe hacer, y no lo que el programa hace. Creemos que es muy importante remarcar que estas estrategias no son opuestas sino complementarias. En nuestra opinión el testing debería estar guiado fundamentalmente por técnicas de MBT pero complementadas con herramientas de análisis de cubrimiento de sentencias de forma tal que los casos generados mediante MBT cubran al menos todas las líneas de código. El término testing funcional lo utilizaremos para designar el testing de la funcionalidad del programa.

# [**Pruebas de Caja Negra y Caja Blanca**](http://ingenierogestion.blogspot.com/2009/06/pruebas-de-caja-negra-y-caja-blanca.html)

De acuerdo al grado de conocimiento de la estructura interna del sistema bajo prueba se pueden clasificar las pruebas en:

* **Pruebas de caja blanca** (White-Box Testing). Son pruebas estructurales. Conociendo el código y siguiendo su estructura lógica, se pueden diseñar pruebas destinadas a comprobar que el código hace correctamente lo que el diseño de bajo nivel indica y otras que demuestren que no se comporta adecuadamente ante determinadas situaciones. Ejemplos típicos de ello son las pruebas unitarias. Se centran en lo que hay codificado o diseñado a bajo nivel por lo que no es necesario conocer la especificación de requisitos, que por otra parte será difícil de relacionar con partes diseñadas a muy bajo nivel.
* **Las pruebas de caja negra** (Black-Box Testing) son pruebas funcionales. Se parte de los requisitos funcionales, a muy alto nivel, para diseñar pruebas que se aplican sobre el sistema sin necesidad de conocer como está construido por dentro (Caja negra). Las pruebas se aplican sobre el sistema empleando un determinado conjunto de datos de entrada y observando las salidas que se producen para determinar si la función se está desempeñando correctamente por el sistema bajo prueba. Las herramientas básicas son observar la funcionalidad y contrastar con la especificación.

Ejemplos típicos de pruebas de caja negra son la comprobación de valores límite, pruebas de integridad de la base de datos, pruebas de situaciones de excepción, o pruebas de rendimiento del sistema. Presentan una limitación en cuanto a que es prácticamente imposible reproducir todo el espectro por la innumerable cantidad de combinaciones de entrada posibles, agravada por el desconocimiento de la lógica interna.

# **Pruebas funcionales**

Una prueba **funcional** es una prueba basada en la ejecución, revisión y retroalimentación de las funcionalidades previamente [diseñadas](https://es.wikipedia.org/wiki/Dise%C3%B1o_orientado_a_objetos) para el [software](https://es.wikipedia.org/wiki/Software). Las pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete [informático](https://es.wikipedia.org/wiki/Inform%C3%A1tica). Dicho de otro modo, son pruebas específicas, concretas y exhaustivas para probar y validar que el software hace lo que debe y, sobre todo, lo que se ha especificado.

**Fases**

Las pruebas funcionales se dividen en las siguientes fases:

### Análisis de requisitos (Planificación)

En esta fase se inicia la elaboración del modelo jerárquico de requisitos de prueba partiendo de los procesos funcionales que soporta el producto o activo de software a evaluar. A partir de las funcionalidades se elaborará el plan de pruebas. Hay que obtener toda la información posible de las aplicaciones sobre las cuales se realizarán las pruebas. Esta información se deberá conseguir de toda la documentación disponible sobre su funcionamiento y hablando con el personal responsable de la misma.

### Diseño del plan de pruebas (Preparación)

En esta fase se identifica, acuerda y especifican los atributos y características de calidad que se van a probar. El objetivo es diseñar las pruebas para que tengan la mayor probabilidad de encontrar defectos con la mínima cantidad de esfuerzo y tiempo. Serán pruebas que se llevarán a cabo a través de la interfaz gráfica del software (GUI). Es decir, demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene. Se crearán casos de prueba divididos en pasos (steps) para cada acción a realizar con un resultado esperado asociado, que podrá ser verificado. Durante la fase de diseño también se especifican los datos de entrada necesarios para que los casos de pruebas definidos puedan ser ejecutados (ya sea buscando el éxito de la prueba, o bien el fallo).

### Ejecución

En esta fase se ejecutarán los casos de prueba anteriormente diseñados de forma manual. Hay que seguir al detalle el guion establecido dejando cierta libertad al tester para detectar situaciones anómalas no contempladas. Las baterías de pruebas serán ejecutadas como mínimo una vez antes del paso a producción, independientemente de las ejecuciones anteriores. Los casos de prueba fallados se reportarán a los desarrolladores para su corrección hasta que su resultado sea correcto.

### Gestión de Incidencias (Defectos)

La gestión de incidencias es una parte implícita de la fase de ejecución, pero que, al tener una alta importancia en las pruebas funcionales, diferenciamos como una etapa independiente. Cuando al realizar la acción de un step el resultado obtenido no es el esperado, habrá que abrir o reportar una incidencia para que el equipo de desarrollo tenga constancia del error. La gestión de incidencias es el principal canal de comunicación con el equipo de desarrollo. Las incidencias han de ser claras y con todo lujo de detalle, tienen que describir el error para que el equipo de desarrollo pueda comprenderlo perfectamente, reproducirlo, localizarlo y poder solucionarlo. Se deberá mantener una continua comunicación con el equipo de desarrollo para conocer el estado de los defectos y poder realizar las repruebas necesarias para su cierre.

## **Según ejecución**

Las pruebas funcionales pueden ser, según su ejecución:

### Manuales

Las pruebas funcionales manuales son las que ejecuta un tester como si fuese un usuario pero siguiendo una serie de pasos establecidos o test plan, diseñado en el análisis de los requisitos para garantizar que hace lo que debe (casos positivos), que no falla (casos negativos) y que es lo que se ha solicitado. El tester realizará las acciones indicadas en cada step del caso de prueba comprobando que se cumple el resultado esperado. Si el resultado es distinto al esperado, se reportará un defecto con todo detalle: descripción, datos utilizados, capturas de pantalla, etc. para facilitar la solución del defecto por parte de los desarrolladores.

### Automáticas

Las pruebas funcionales automáticas son pruebas funcionales que se automatizan para "ahorrar tiempo de pruebas". A partir de los casos de prueba de las pruebas manuales, se automatizan los casos de prueba que se repitan en las ejecuciones. Esos casos suelen ser los más importantes (happy flow) de los módulos o procesos de negocio "vitales" de la aplicación, es decir, los procesos que siempre tienen que funcionar y que bajo ningún concepto pueden fallar. El objetivo de las pruebas funcionales automáticas es comprobar que nada de lo probado con anterioridad ha dejado de funcionar como debería.

## **Según tipo de pruebas**

Las pruebas funcionales pueden ser, según el nivel y el tipo de pruebas:

### Pruebas exploratorias

Son aquellas pruebas a través de las cuales, simultáneamente, se obtiene un aprendizaje y conocimiento de la aplicación a probar a la vez que se genera un valor desde el primer momento. Ayudan a la integración de la fase de pruebas de una forma mucho más rápida, pues permiten al tester elaborar un guion de pruebas que utilizará para el diseño de los futuros planes de pruebas. Estas pruebas son realmente útiles a la hora de probar aplicaciones ya desarrolladas, es decir, aquellas pruebas de software que no comienzan a la vez que el desarrollo. Para realizar las pruebas funcionales exploratorias se identificarán los distintos procesos de negocio o módulos de la aplicación y se le dará al tester libertad, poniéndose en la piel de un usuario, para probarlos. Estas pruebas exploratorias deberán ejecutarse sobre la última versión cerrada disponible de la aplicación.

### Pruebas de regresión

*Artículo principal:*[Pruebas de regresión](https://es.wikipedia.org/wiki/Pruebas_de_regresi%C3%B3n)

El objetivo de las pruebas de regresión es eliminar el efecto onda, es decir, comprobar que cambios realizados en el software no introducen un comportamiento no deseado o errores adicionales en otros módulos o partes no modificados. Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error como para realizar una mejora. No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes. Este tipo de pruebas tiene que garantizar que, tras un cambio en el software, al menos la funcionalidad más importante sigue funcionando. Para este tipo de pruebas lo ideal es automatizar los casos que validen que estas partes siguen funcionando, pues se ejecutarán de manera repetitiva a lo largo del ciclo de vida del software.

### Pruebas de compatibilidad

*Artículo principal:*[Pruebas de compatibilidad](https://es.wikipedia.org/wiki/Pruebas_de_compatibilidad)

Las pruebas de compatibilidad son pruebas funcionales realizadas en diferentes entornos como en cada navegador de internet, sistema operativo o dispositivo, para garantizar el correcto funcionamiento de la aplicación en todos los medios. El mismo software puede presentar errores dependiendo de dónde se ejecute: funcionales (botones y enlaces pueden dejar de funcionar, producen errores de sistema o simplemente no realizan la funcionalidad esperada), estéticos (pueden descuadrarse frames de la aplicación, no cargarse imágenes, desaparecer enlaces o botones y textos).

### Pruebas de integración

*Artículo principal:*[Pruebas de integración](https://es.wikipedia.org/wiki/Pruebas_de_integraci%C3%B3n)

Las pruebas de integración son pruebas funcionales entre dos o más sistemas. El objetivo de las pruebas de integración es verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos.

### Pruebas de aceptación

*Artículo principal:*[Pruebas de aceptación (informática)](https://es.wikipedia.org/wiki/Pruebas_de_aceptaci%C3%B3n_(inform%C3%A1tica))

El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema que determine su aceptación, desde el punto de vista de su funcionalidad y rendimiento. En las pruebas de aceptación, la ejecución y aprobación final corresponden al usuario o cliente, que es el que valida y verifica que el alcance es el correcto.

## **Metodologías**

### La metodología estándar o en cascada

Es una metodología lineal. Se ordenan rigurosamente las etapas del proceso de tal forma que el inicio de cada etapa debe esperar a la finalización de la etapa anterior. El equipo de pruebas trabaja de forma paralela al equipo de desarrollo y empieza a ejecutar o pasar las pruebas una vez el desarrollo está completado.

### Las metodologías ágiles

Se basan en el desarrollo iterativo e incremental, donde los requerimientos y soluciones evolucionan mediante la colaboración de grupos auto-organizados y multidisciplinarios, minimizando riesgos desarrollando en lapsos cortos de tiempo. Estos lapsos se denominan "Iteración". Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. Una iteración no debe agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, pero la meta es tener algo tangible (sin errores) al final de cada iteración. Al final de cada iteración el equipo vuelve a evaluar las prioridades del proyecto.

# **Desarrollo en cascada**

En Ingeniería de *software* el **desarrollo en cascada**, también llamado **modelo en cascada** (denominado así por la posición de las fases en el desarrollo de esta, que parecen caer en cascada *“por gravedad”* hacia las siguientes fases), es el enfoque metodológico que ordena rigurosamente las etapas del **proceso para el desarrollo de *software***, de tal forma que el inicio de cada etapa debe esperar a la finalización de la etapa anterior.[1](https://es.wikipedia.org/wiki/Desarrollo_en_cascada#cite_note-1)​ Al final de cada etapa, el modelo está diseñado para llevar a cabo una revisión final, que se encarga de determinar si el proyecto está listo para avanzar a la siguiente fase. Este modelo fue el primero en originarse y es la base de todos los demás modelos de ciclo de vida....

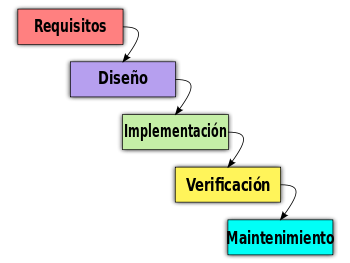
La versión original fue propuesta por Winston W. Royce en 1970 y posteriormente revisada por Barry Boehm en 1980 e Ian Sommerville en 1985.[2](https://es.wikipedia.org/wiki/Desarrollo_en_cascada#cite_note-te-2)​

Un ejemplo de una metodología de desarrollo en cascada es:

1. Análisis de requisitos.
2. Diseño del sistema.
3. Diseño del programa.
4. Codificación.
5. Pruebas.
6. Implementación del programa.
7. Mantenimiento.

De esta forma, cualquier error de diseño detectado en la etapa de prueba conduce necesariamente al rediseño y nueva programación del código afectado, aumentando los costos del desarrollo. La palabra *cascada* sugiere, mediante la metáfora de la fuerza de la gravedad, el esfuerzo necesario para introducir un cambio en las fases más avanzadas de un proyecto.

## **Fases del modelo**

[](https://commons.wikimedia.org/wiki/File:El_modelo_de_desarrollo_en_cascada.svg)

El "modelo cascada" sin modificar. El progreso fluye de arriba hacia abajo, como una cascada.

### Análisis de requisitos del software

En esta fase se analizan las necesidades de los usuarios finales del *software* para determinar qué objetivos debe cubrir. De esta fase surge una memoria llamada SRD (documento de especificación de requisitos), que contiene la especificación completa de lo que debe hacer el sistema sin entrar en detalles internos.

Es importante señalar que en esta etapa se debe **consensuar** todo lo que se requiere del sistema y será aquello lo que seguirá en las siguientes etapas, no pudiéndose requerir nuevos resultados a mitad del proceso de elaboración del *software* de una manera.

### Diseño del sistema

Descompone y organiza el sistema en elementos que puedan elaborarse por separado, aprovechando las ventajas del desarrollo en equipo. Como resultado surge el SDD (Documento de Diseño del *Software*), que contiene la descripción de la estructura relacional global del sistema y la especificación de lo que debe hacer cada una de sus partes, así como la manera en que se combinan unas con otras.

Es conveniente distinguir entre diseño de alto nivel o arquitectónico y diseño detallado. El primero de ellos tiene como objetivo definir la estructura de la solución (una vez que la fase de análisis ha descrito el problema) identificando grandes módulos (conjuntos de funciones que van a estar asociadas) y sus relaciones. Con ello se define la arquitectura de la solución elegida. El segundo define los algoritmos empleados y la organización del código para comenzar la implementación.

### Diseño del programa

Es la fase en donde se realizan los algoritmos necesarios para el cumplimiento de los requerimientos del usuario, así como también los análisis necesarios para saber qué herramientas usar en la etapa de Codificación.

### Codificación

Es la fase en donde se implementa el código fuente, haciendo uso de prototipos, así como de pruebas y ensayos para corregir errores. Dependiendo del lenguaje de programación y su versión se crean las bibliotecas y componentes reutilizables dentro del mismo proyecto para hacer que la programación sea un proceso mucho más rápido.

### Pruebas

Los elementos, ya programados, se ensamblan para componer el sistema y se comprueba que funciona correctamente y que cumple con los requisitos, antes de ser entregado al usuario final.

### Verificación

Es la fase en donde el usuario final ejecuta el sistema, para ello el o los programadores ya realizaron exhaustivas pruebas para comprobar que el sistema no falle.

### Mantenimiento

Una de las etapas más críticas, ya que se destina un 75 % de los recursos, es el mantenimiento del *software* ya que al utilizarlo como usuario final puede ser que no cumpla con todas nuestras expectativas.

# **Kanban (desarrollo)**

Este artículo se refiere a la gestión de procesos y método de mejora. Para el proceso de fabricación optimizada "lean manufacturing", consulte [Kanban](https://es.wikipedia.org/wiki/Kanban" \o "Kanban).

**Kanban** es un método para gestionar el trabajo intelectual, con énfasis en la entrega justo a tiempo, mientras no se sobrecarguen los miembros del equipo. En este enfoque, el proceso, desde la definición de una tarea hasta su entrega al cliente, se muestra para que los participantes lo vean y los miembros del equipo tomen el trabajo de una cola.

Kanban se puede dividir en dos partes:

* Kanban - Un sistema de gestión de proceso visual que le indica qué producir, cuándo producirlo, y cuánto producir.
* El método Kanban - Una aproximación a la mejora del proceso evolutivo e incremental para las organizaciones.

## **El método Kanban**

En el desarrollo de software, se utiliza el sistema Kanban virtual para limitar el trabajo en curso. A pesar de que el nombre se origina del idioma japonés "Kanban", y se traduce aproximadamente como "tarjeta de señal", y hay tarjetas utilizadas en la mayoría de las implementaciones de Kanban en desarrollo de software, estas tarjetas no funcionan en realidad como señales para realizar más trabajo. Representan los elementos de trabajo. De ahí el término "virtual" porque no existe una tarjeta física.

El método Kanban formulado por David J. Anderson[1](https://es.wikipedia.org/wiki/Kanban_(desarrollo)#cite_note-1)​[2](https://es.wikipedia.org/wiki/Kanban_(desarrollo)#cite_note-2)​ es una aproximación al proceso gradual, evolutivo y al cambio de sistemas para las organizaciones. Utiliza un sistema de extracción limitada del trabajo en curso como mecanismo básico para exponer los problemas de funcionamiento del sistema (o proceso) y estimular la colaboración para la mejora continua del sistema. Un ejemplo del sistema de extracción es el sistema Kanban, y es después de esta popular forma de trabajo en curso, que se ha denominado el método.

## **Los principios del método Kanban**

El método Kanban tiene sus raíces en cuatro principios básicos:

1. Comience con lo que hace ahora

El método Kanban se inicia con las funciones y procesos que ya se tienen y estimula cambios continuos, incrementales y evolutivos a su sistema.

1. Se acuerda perseguir el cambio incremental y evolutivo

La organización (o equipo) deben estar de acuerdo que el cambio continuo, gradual y evolutivo es la manera de hacer mejoras en el sistema y debe apegarse a ello. Los cambios radicales pueden parecer más eficaces, pero tienen una mayor tasa de fracaso debido a la resistencia y el miedo en la organización. El método Kanban anima a los pequeños y continuos cambios incrementales y evolutivos a su sistema actual.

1. Respetar el proceso actual, los roles, las responsabilidades y los cargos

Tenemos que facilitar el cambio futuro; acordando respetar los roles actuales, responsabilidades y cargos, eliminamos los temores iniciales. Esto nos debería permitir obtener un mayor apoyo a nuestra iniciativa Kanban.

1. Liderazgo en todos los niveles

Se debe alentar hechos de liderazgo en todos los niveles de la organización de los contribuyentes individuales a la alta dirección.

## **Cinco prácticas centrales del método Kanban**

Anderson identificó cinco características básicas que habían sido observadas en cada implementación correcta del método Kanban. Posteriormente fueron etiquetadas como prácticas y se ampliaron con la adición de una sexta característica.[3](https://es.wikipedia.org/wiki/Kanban_(desarrollo)#cite_note-3)​

1. Visualizar

Visualizar el flujo de trabajo y hacerlo visible es la base para comprender cómo avanza el trabajo. Sin comprender el flujo de trabajo, realizar los cambios adecuados es más difícil. Una forma común de visualizar el flujo de trabajo es el uso de columnas. Las columnas representan los diferentes estados o pasos en el flujo de trabajo.

1. Limitar el trabajo en curso

Limitar el trabajo en curso implica que un sistema de extracción se aplica en la totalidad o parte del flujo de trabajo. El sistema de extracción actúa como uno de los principales estímulos para los cambios continuos, incrementales y evolutivos en el sistema.

1. Dirigir y gestionar el flujo

Se debe supervisar, medir y reportar el flujo de trabajo a través de cada estado. Al gestionar activamente el flujo, los cambios continuos, graduales y evolutivos del sistema pueden ser evaluados para tener efectos positivos o negativos.

1. Hacer las Políticas de Proceso Explícitas

Configure las reglas y directrices de su trabajo. Entienda las necesidades y asegúrese de seguir las reglas. Las políticas definirán cuándo y por qué una tarjeta debe pasar de una columna a otra. Escríbalas. Cambie las reglas cuando la realidad cambie.

1. Utilizar modelos para reconocer oportunidades de mejora

Cuando los equipos tienen un entendimiento común de las teorías sobre el trabajo, el flujo de trabajo, el proceso y el riesgo, es más probable que sea capaz de construir una comprensión compartida de un problema y proponer acciones de mejora que puedan ser aprobadas por consenso. El método Kanban sugiere que un enfoque científico sea utilizado para implementar los cambios continuos, graduales y evolutivos. El método no prescribe un método científico específico para utilizarlo.

## **Falla - Error - Defecto**This is a featured page

Es común ver como algunas personas utilizan de manera indistinta los términos Defecto, Falla y Error. Sin embargo, cada uno de ellos tiene un significado diferente:

* **Defecto:** Un defecto se encuentra en un artefacto y puede definirse como una diferencia entre la versión correcta del artefacto y una versión incorrecta. Coincide con la definición de diccionario, **"imperfección"**.
* **Falla:** En terminología IEEE, una falla es la **discrepancia visible** que se produce al ejecutar un programa con un defecto, el cual es incapaz de funcionar correctamente (no sigue su curso normal).
* **Error:** Es una equivocación cometida por el desarrollador. Algunos ejemplos de errores son: un error de digitación, una malinterpretación de un requerimiento o de la funcionalidad de un método. El estándar 829 de la IEEE coincide con la definición de diccionario de error como **"una idea falsa o equivocada".** Por tal razón un programa no puede tener o estar en un error, ya que los programas no tienen ideas; las ideas las tienen la gente.

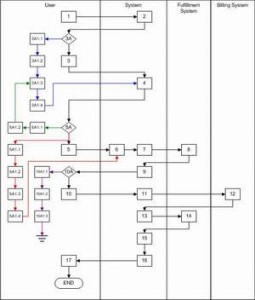
Ampliando el tema de los errores, estos errores ocurren cuando cualquier aspecto de un producto de software es incompleto, inconsistente o incorrecto. Las tres grandes clases clasificaciones de errores del software son los de **requisitos,** de **diseño**y de **implantación**. En esta sección hablaremos de dos de ellos: 

* Errores de diseño: Se introducen por fallas al traducir los requisitos en estructuras de solución correctas y completas, por inconsistencias tanto dentro de las especificaciones de diseño y como entre las especificaciones de diseño y los requisitos. Un error de requisitos o un error de diseño, que no se descubre sino hasta las pruebas de código fuente, puede ser muy costoso de corregir. De modo que es importante que la calidad de los requisitos y de los documentos del diseño se valoren pronto y con frecuencia.
* Los errores de instrumentación: Son los cometidos al traducir las especificaciones de diseño en código fuente. Estos errores pueden ocurrir en las declaraciones de datos, en las referencias a los datos, en la lógica del flujo de control, en expresiones computacionales, en interfaces entre subprogramas y en operaciones de entrada/salida. La calidad de los productos de trabajo generados durante el análisis y el diseño se puede estimar y mejorar utilizando procedimientos sistemáticos de control de calidad, mediante recorridos e inspecciones y por medio de verificaciones automatizadas para supervisar que sea consistente y que esté completo. Las técnicas para estimar y mejorar la calidad del código fuente incluyen los procedimientos sistemáticos de control de calidad, los recorridos inspecciones, el análisis estático, la ejecución simbólica, las pruebas de unidad y las pruebas de integración sistemáticas. Las técnicas de verificación formal se pueden usar para mostrar, de manera rigurosa, que un programa fuente se conforma con sus requisitos; la verificación formal también puede servir para guiar la síntesis sistemática de los programas fuente.

## **¿Diferencia entre un caso de uso y un caso de prueba?**

### Definición de Caso de Uso

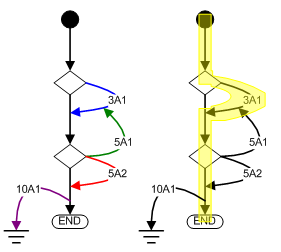
Los **casos de uso** cuentan la historia de cómo una persona interactúa con un sistema de software para lograr un objetivo. Un buen **caso de uso** describe las interacciones que conducen a lograr o abandonar el objetivo. En el **caso de uso** se describen múltiples rutas que el usuario puede seguir en el caso de uso.

**[](https://testeandosoftware.com/wp-content/uploads/2013/09/full-flow.jpg)**

Caso de Uso

### Definición de Escenario de Caso de Uso

Un caso de uso se compone de uno o más **escenarios de casos de uso**. Cada camino que se puede seguir en el caso de uso es un escenario de caso de uso. Cualquier ejemplo que se da a raíz de un caso de uso también sigue un solo escenario. Múltiples ejecuciones del caso de uso se pueden usar los mismos o diferentes escenarios.

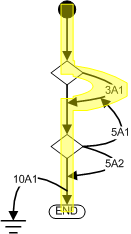
**[](https://testeandosoftware.com/wp-content/uploads/2013/09/UseCaseScenario.png)**

El recorrido marcado en amarillo sería un Escenario de Caso de Uso.

Dicho en otras palabras, un escenario de caso de uso es una instancia del caso de uso que describe una ruta específica a través del flujo de eventos.

### Definición de un caso de prueba

Un **caso de prueba cubre el software más a fondo y con más detalle que un caso de uso**. Los **casos de prueba** incluyen todas las funciones que el programa es capaz de realizar (o se supone que es capaz de realizar). Los **casos de prueba** deben tener en cuenta el uso de todo tipo de datos de entrada/salida, cada comportamiento esperado, todos los elementos de diseño, y cada clase de defecto. Hay una gran cantidad de requisitos que no están cubiertos en los casos de uso. Sin embargo, todos los requisitos deberán ser cubiertos por los casos de prueba.

**[](https://testeandosoftware.com/wp-content/uploads/2013/09/flow-uncolored-sequence.png)**

A partir del escenario de caso de uso crearemos los Casos de Prueba, que son la la descripción de las actividades que se van a ejecutar con el fin de validar la aplicación.

Dicho esto, hay que decir que hay muchos tipos diferentes de pruebas, que se pueden describir de diferentes maneras. Estas pruebas pueden ser realizadas por diferentes personas para lograr diferentes objetivos. Las pruebas pueden ser manuales o automatizadas. Todo esto genera diferentes casos de prueba.

Por ejemplo, se pueden crear **dos casos de prueba** diseñadas **para comprobar** que se cumplen los requisitos de **un solo caso de uso**. El primer caso de prueba implicaría un usuario que tiene una cuenta, pero elige usar información diferente para realizar su pedido. Este es un escenario realista – tal vez alguien está pidiendo un regalo para ser entregado a su primo que vive en una dirección diferente. También se puede definir un sistema de prueba que consiste en un usuario sin una cuenta pre-existente.

Los dos casos de prueba siguen un caso de uso, y los dos casos de prueba siguen el escenario de caso de uso. Sin embargo, los casos de prueba prueban cosas diferentes (desde un punto de vista de requisitos). Pueden estar probando exactamente el mismo código, pero desde una perspectiva de prueba del sistema, que no saben ni les importa porque una prueba del sistema es una prueba de caja negra.

### En resumen

Un **caso de uso** representa la interacción (o conductas observadas) asociados con el logro o el abandono de una meta. Un **escenario de caso** de uso representa uno de los posibles caminos a través de un caso de uso. Un **caso de prueba** representa un conjunto de entradas de datos (actividades) que ejecutan un solo escenario de caso de uso para obtener unos datos de salida esperados (resultados).

## [**Casos de Prueba y/o Escenarios**](https://losandestraining.com/2017/10/19/casos-de-prueba-yo-escenarios/)

No existe una fórmula que funciona para todos los proyectos o en este caos para todas las aplicaciones, si bien hoy en día la mayoría de los equipos usan scrum en su proceso de desarrollo se mal interpreta al pensar que al ser una metodología ágil no es necesario escribir test cases, si bien las aplicaciones en el aspecto de desarrollo avanzan rápidamente ya que al final de cada sprint tenemos un incremento en la aplicación, es mucho más importante asegurar la calidad de ese incremento. También es una realidad que los tiempos para crear test cases, automatizar y probar son cada vez menos, no podemos dejar de usar las herramientas que tenemos para asegurar calidad. ¿La pregunta es creamos test cases o no? Si, toda herramienta que nos ayude a obtener un producto de calidad debe usarse, pero como dijimos antes no es necesario crear los test cases de manera tradicional.

### ¿Cuál es la diferencia entre un test case y escenarios?

Un test case es un conjunto de condiciones bajo las cuales se determina que una parte determinada de una aplicación funciona de acuerdo a los requerimientos, y para escribir un test case debemos tener precondiciones, paso a reproducir, resultado esperado, datos de prueba, etc. En cambio, para escribir un escenario solo necesitamos saber que debe realizar la aplicación.

Tal vez una aproximación para entender la diferencia entre un test case y un escenario sería:

Test Case: Como debe probarse.

Escenario: Que se deber probar.

Después de dar una breve definición de los test cases, veamos la diferencia con un escenario. Los escenarios solo describen que debe hacer la aplicación, y pueden ser escritos en una línea o más y ser parte de un conjunto de escenarios como un checklist.

### ¿Cuándo usar escenarios?

Cuando las reglas de negocio son complejas

Es más fácil pensar en el resultado esperando a un determinado flujo de acciones, que tener que escribir paso a paso y además es más efectivo en cuestiones de tiempo.

Cuando el sistema cambia de un momento a otro

Mantener los test cases involucra tiempo, durante el cual a veces esos mismos test case ya no útiles para el estado actual de la aplicación.

Cuando las diferentes combinaciones para probar una determinada aplicación son muchas

En este caso escribir la cantidad de test cases para probar una sola parte de la aplicación involucra mucho tiempo y nivel de detalle para diferenciar un test cases de otro.

Cuando no se puede reutilizar los test cases

Cuando se tiene por ejemplo una aplicación que interactúa con otras aplicaciones, pero, los pasos para ejecutar determinado test casen no son los mismo, lo único que es igual es el resultado final.

### Que se debe tener en cuenta a la hora de escribir escenarios

Entender el sistema/aplicación en la cual se está trabajando. EL propósito de la aplicación.

Identificar los escenarios más comunes

Pensar como el usuario final, el usuario final no piensa en paso o precondiciones, solo en resultados.