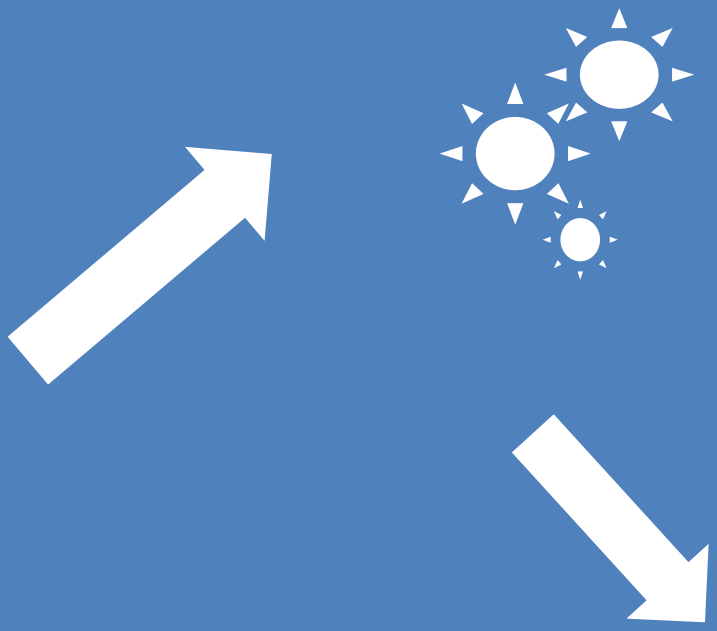


INTRODUCCIÓN AL SOFTWARE TESTING

Luis González Varela. Validación y pruebas. 2012.

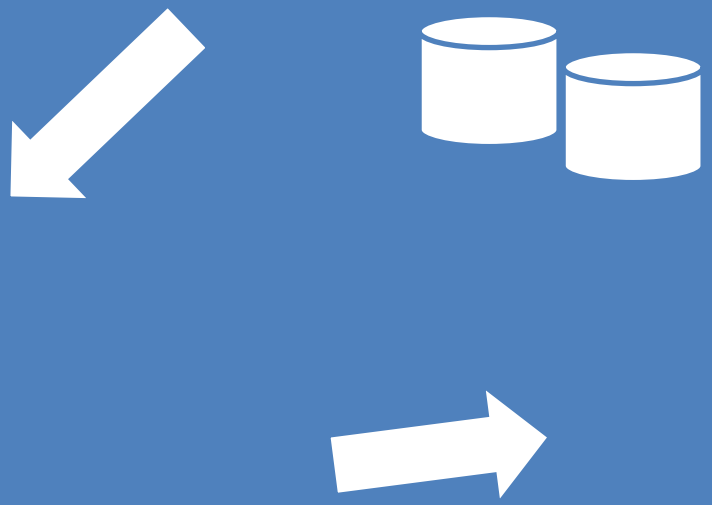


Tabla de contenido.

Introducción.

Las pruebas son el único instrumento que puede determinar la calidad de un producto software, es decir, es el único método por el que se puede asegurar que un sistema software cumple con los requerimientos.

Se puede determinar una dependencia directa entre la calidad de un sistema y el valor del mismo, por lo tanto, las pruebas dotan de valor a los productos *software*.

Se estima que el coste de las pruebas, en un desarrollo comercial básico, puede suponer entre un 50% y un 75% del coste total del proyecto. Pero, ¿Si son tan caras por qué deben aplicarse?

Aunque a efectos totales las pruebas suponen un coste importante, tanto en tiempo como en recursos asociados, aplicarlas reduce todos los demás costes, es decir, los costes de desarrollo, implementación y mantenimiento se ven enormemente reducidos al tener un sistema probado. Al mismo tiempo las pruebas sirven como documentos, contratos, que aseguran el funcionamiento del sistema dotándolo de una fiabilidad probada.

Pero no sólo hay que aplicar pruebas a un producto software, hay que hacerlo de una manera controlada y correcta. El uso inadecuado de pruebas tiene como consecuencia: baja calidad, aumento de los costes de desarrollo, retraso en la comercialización/publicación, aumento de los costes de transacciones mercantiles, etc.

Para probar de una forma correcta los productos software surgen diferentes metodologías a lo largo del tiempo. Metodologías que se adaptan evolucionando de forma paralela con la Ingeniería del software.

Un poco de historia.

Las pruebas ubicadas dentro de la historia de la Ingeniería del software y siendo organizadas por un criterio de orientación, o paradigma, que permita hacer un balance sobre su evolución pueden subdividirse, por el momento, en cinco intervalos temporales o fases:

1. **Antes de 1956. Periodo orientado a *debugging*.**
2. **Entre 1957 y 1978. Periodo orientado a demostración.**
3. **Entre 1979 y 1982. Periodo orientado a destrucción.**
4. **Entre 1983 y 1984. Periodo orientado a evaluación.**
5. **Entre 1985 y la actualidad. Periodo orientado a prevención.**

Fase 1. Antes de 1956. Periodo orientado a *debugging*.

En este momento todas **las pruebas que se realizaban estaban orientadas a la corrección directa del código** fuente de los programas. Eran realizadas directamente por los programadores y no estaba clara la diferencia entre: *checkout*, *debugging* y *testing*.

El concepto de test o prueba en lo referente al software parte, o es precursor del mismo, *A.M. Turing* que en 1949 redacta: *Checking a Large Routine* donde ya se discute el uso de aserciones para realizar pruebas de corrección.

Fase 2. Entre 1957 y 1978. Periodo orientado a demostración.

En este momento las pruebas se centran en la realización de *checkouts* exhaustivos que se focalizan en dos aspectos clave. Por un lado **asegurar que el programa funciona** (*Debugging*) **y** por otro **asegurar que el programa resuelve el problema** (*Testing*), tal y como expone en 1957 C. Baker en su artículo *Review of D.D. McCracken's Digital Computer Programming.*

Podría afirmarse que en esta fase se utilizan de forma masiva test para garantizar que se cumple con la especificación. Dichos test se realizan al final del desarrollo del software.

"La prueba de software puede demostrar la existencia de fallos, pero nunca podría demostrar la ausencia de los mismos." *E.W.Dijkstra.*

Fase 3. Entre 1979 y 1982. Periodo orientado a destrucción.

En 1979 *G.J. Myers,* publica *The Art of Software Testing*, donde expone que el "*Testing* es el proceso de ejecutar un programa con la intención de encontrar errores."

Esto cambia por completo el paradigma de las pruebas ya que se pasa de intentar **demostrar que un programa es correcto** mediante pruebas y demostraciones teóricas basadas en matemáticas a intentar hacer fallar el programa. El objetivo no cambia, intentar que el programa no tenga fallos, pero si la forma de buscarlos y garantizarlos.

En este momento si una prueba produce un fallo se considera que la prueba ha tenido éxito. A efectos de concepto, los *tests* se convierten en Casos de Prueba que se aplican a los productos desarrollados para encontrar errores y corregirlos.

"Las pruebas de software son el proceso de ejecutar un programa con la intención de encontrar errores" *G.J. Myers.*

Fase 4. Entre 1983 y 1984. Periodo orientado a evaluación.

En 1983 *Guideline for Lifecycle Validation, Verification and Testing of Computer Software*. *NBS FIPS* propone y describe una metodología que integra análisis, revisión y *testing* en el ciclo de vida del desarrollo de software.

Las pruebas de Software empiezan **a integrarse en las diferentes metodologías de desarrollo de software**.

"El objetivo general de las pruebas de software es confirmar la calidad de los sistemas software ejercitando sistemáticamente el software en unas circunstancias cuidadosamente controladas" *E.F. Miller*.

Fase 5. Entre 1985 en adelante. Periodo orientado a prevención.

En 1985 *H. Hetzel,* y *D. Gelperin,* implementan STEP (*Systematic Test and Evaluation Process*) sobre los estándares formales IEEE 829-1983 y 828-1998. Este hecho consigue que las pruebas del software cobren una mayor importancia en el ciclo de desarrollo de software con lo que se abre la posibilidad de integrar pruebas en las diferentes fases del mismo.

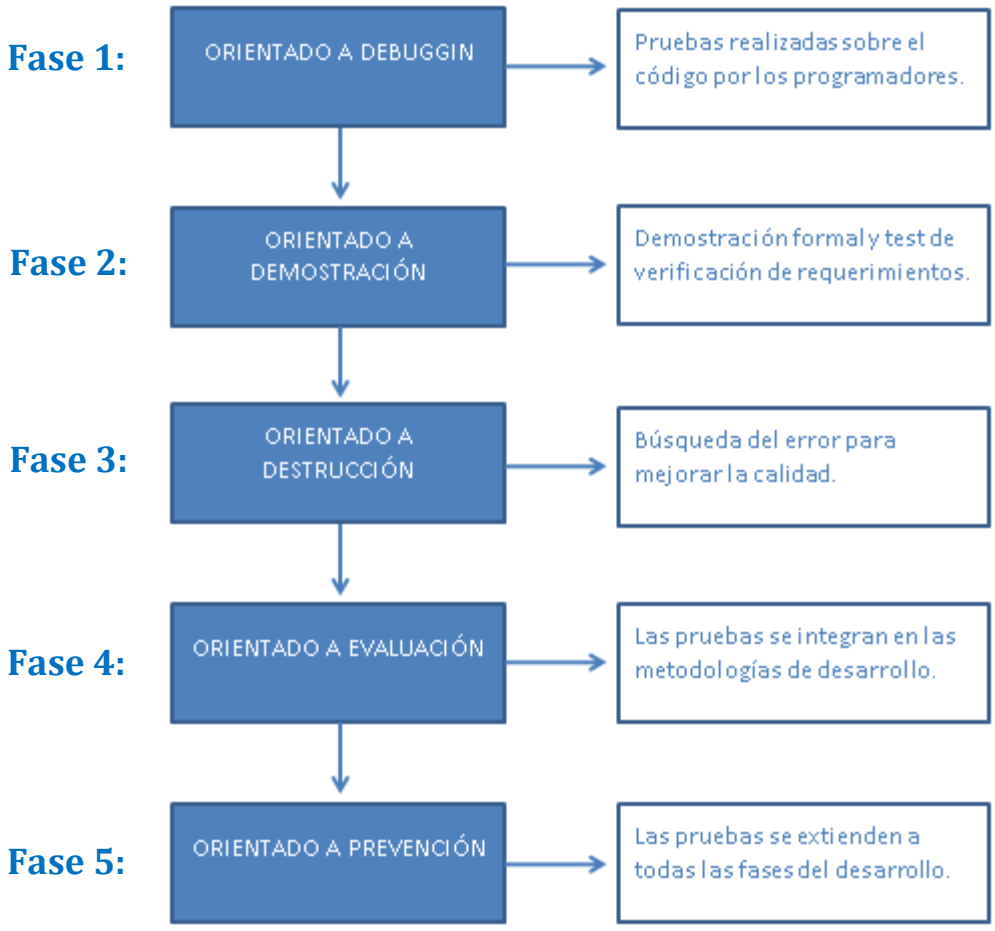
Durante esta época las pruebas se diversifican para cubrir todas las fases del desarrollo, poder comprobar todos los tipos de artefactos, prototipos, modelos, módulos, subsistemas y sistemas que componen los productos software del momento.

Poco a poco las pruebas se considerarán **una parte clave de todo el ciclo de desarrollo**.

Será en 1999 cuando se establezcan las bases de la Programación Extrema (*Kent Beck*) de las cuales derivará el *Test Driven Development,* una metodología de desarrollo basada en pruebas.

"Hacer pruebas significa comparar los resultados actuales con un estándar" *Hutcheson, M.L.*

Evolución del *Software Testing*:



Definiendo el *Software Testing*.

El reto de *Software Testing.*

El *Software Testing* es un proceso de la Ingeniería por el cual se controla que un sistema cumpla con sus requisitos funcionales y funcione correctamente adecuándolo a unos estándares de calidad y fiabilidad.

Uno de los problemas principales del software radica en la dificultad de determinar el origen de un error concreto.

Generalmente las causas de un error se ubican dentro de un conjunto predefinido de posibilidades:

1. **El usuario ejecuta código no testeado.**

   Esto puede deberse a posibles restricciones de tiempo en el desarrollo de software, a que las pruebas se hayan considera secundarias y no se hayan realizado hasta el final o a que estén mal planteadas y no cubran todos los escenarios necesarios para garantizar un mínimo de calidad.

   De este conjunto se puede deducir que el tiempo necesario para plantear todos los posibles escenarios necesarios para garantizar que un sistema no falla o por lo menos para cubrir los mínimos necesarios de efectividad es bastante amplio ya que la cobertura de un sistema depende de forma directamente proporcional de la complejidad del mismo.

2. **El orden en el que se ejecutaron las sentencias no es el mismo que el orden que se ha probado**.

   Es probable que no se hayan cubierto todas las posibilidades combinacionales posibles. Sistemas recursivos, inteligencia artificial, interfaces de usuario complejos, etc.

3. **El usuario ha introducido una combinación de valores de entrada no testeados.**

   Por definición es inviable testear todas las posibilidades de entrada de un sistema. Valorando únicamente una entrada numérica nos encontramos ante infinitos valores posibles por lo que asegurar de forma precisa que un sistema funcionará correctamente con todas las posibles entradas es, a efectos prácticos, imposible.

Los encargados de estructurar las pruebas deberán decidir que casos deben cubrir y cuales no.

Todo lo relacionado con el usuario exige una gran cantidad de trabajo en lo referente al control de las posibilidades ya que los usuarios son impredecibles.

En el siguiente ejemplo podemos observar como aumentan las líneas de código para controlar las posibles entradas de un usuario.

```
#include <stdio.h>
#include <conio.h>

main () {
    int x,y;
    printf ("Introduce el valor de x ");
    scanf ("%d",&x);
    printf ("\nIntroduce el valor y ");
    scanf ("%d",&y);
    printf ("\nNumeros introducidos: %d, %d",x,y);

    if ((x >= 0) && (y >= 0))
        printf ("\nLa suma de los numeros es: %d",x+y);
    else
        if ((x<0) && (y<0))
            printf ("\nNo se calcula porque los dos numeros son negativos");
        else
            if (x<0)
                printf ("\nNo se calcula la suma porque el primer numero es negativo");
            else
                printf ("\nNo se calcula la suma porque el segundo numero es negativo");

    getch();
}
```

4. **El entorno operativo del usuario nunca ha sido testeado.**

   La expansión de los sistemas operativos y de los diferentes navegadores funcionales para internet ha diversificado las características internas y por lo tanto de los requisitos no funcionales de todo desarrollo.

   Es probable que una aplicación concreta desarrollada en un lenguaje concreto y sobre unas librerías concretas no funcione correctamente en todos los sistemas operativos y en todas sus versiones.

Fases del *Software Testing.*

El proceso de prueba del software puede dividirse en cuatro fases:

1. **Modelar el entorno del software.**

   Modelar el entorno del software consiste en: identificar y emular las interfaces que usa el sistema de software, identificar y enumerar las entradas del sistema, definir las interfaces (humanas, software, sistema de ficheros, protocolos, drivers, etc.) que usará nuestro sistema y determinar que entradas y que secuencias de las mismas deben ser probadas.

   Durante el modelado del entorno pueden surgir problemas en lo referente a la amplitud y la complejidad de las pruebas. Por ello los encargados de definir las pruebas deberán determinar que valores de entrada serán probados y que secuencias de valores es necesario probar y cuales no.

   El criterio de selección dependerá del tiempo disponible, el número de recursos disponibles, las especificaciones de requerimientos funcionales y no funcionales y la complejidad de elaboración de cada prueba.

2. **Seleccionar los escenarios de prueba.**

   El número de escenarios posibles suele ser infinito y cada uno de ellos tendrá un coste determinado asociado y un tiempo para evaluarse. Como es evidente, en un desarrollo sólo se podrán evaluar un número finito de escenarios surgiendo la cuestión de ¿qué escenarios deben evaluarse y cuáles no?

   Al mismo tiempo, las secuencias de entrada, es decir, las combinaciones de las entradas pueden ser infinitas, por ejemplo un entorno gráfico o interfaz muy complejo donde existen cientos de opciones que a su vez contienen múltiples opciones de interactuación. Determinar que secuencias son necesarias y cuales no también es un problema.

3. **Ejecutar y evaluar.**

   Ejecutar y evaluar los escenarios consiste en probarlos bien manualmente o de forma automatizada. Estamos ante una tarea tediosa e intensiva donde hay una gran posibilidad de cometer errores por parte de los probadores.

De igual manera la necesidad de simular las entradas y los destinos de las salidas o parte del sistema si éste no se ha construido aún puede ser una tarea complicada e inducir errores en los resultados de las pruebas.

Por otro lado cuando se detecta un error y se "arregla", la tendencia es que surjan nuevos errores derivados o que el primero había ocultado.

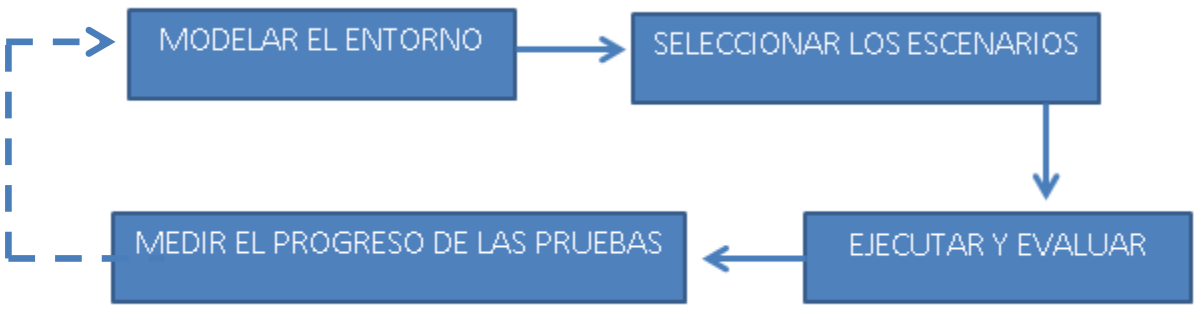
4. **Medir el progreso de las pruebas.**

La evaluación de las pruebas es una combinación de luces y oscuridad sin remedio aparente. ¿Encontrar fallos es bueno o malo?

Por un lado es bueno porque si encontramos errores y los solucionamos nuestro sistema mejora y contiene menos errores. Por el otro lado es malo porque si nuestro sistema contenía muchos errores lo más probable es que siga conteniendo más.

Esta paradoja hace que nos preguntemos si es posible evaluar la fiabilidad del software. Lo cierto es que no existe un método formal que defina si es bueno o malo la detección de errores. La práctica nos indica que un módulo concreto en el que se hayan detectado muchos errores tiende a contener muchos más o lo que es lo mismo, es de baja calidad pero no encontrar errores o encontrar pocos no quiere decir que no existan por lo que la calidad siempre será relativa.

Fases del *Software Testing*:



Introducción a los Tipos de Pruebas.

Existen múltiples criterios y formas de clasificación para las pruebas dependiendo del ámbito concreto en el que se realice dicha clasificación:

- **Validación vs Verificación.**

  La Validación es un **proceso subjetivo** donde se valora si es correcto lo **que** hace un sistema, es decir, **se centra en el resultado**.

  La Verificación es un **proceso objetivo** que se centra en lo que se supone que debe hacer un programa, comprobando que todo se **ejecuta de forma correcta y que no se ejecutan funciones que puedan degradar el funcionamiento**.

- **Formales vs No Formales.**

  Las pruebas Formales necesitan **una especificación formal del problema** y por ello suelen ser muy potentes. Dicha especificación es **costosa y compleja** y suele requerir personal cualificado. Este tipo de pruebas suele aplicarse en ámbitos muy concretos, es decir**, zonas críticas del sistema**.

  Las pruebas No-Formales se basan en especificaciones realizadas en lenguaje natural (**Historias de usuario**), son menos potentes pero su **uso es generalizado**.

- **Estáticas vs Dinámicas.**

  Las pruebas Estáticas **no requieren ejecución del programa**, normalmente se basan en la **revisión del código fuente** y en la creación de trazas de ejecución. Suelen ser manuales.

  Las pruebas Dinámicas **requieren la ejecución del programa**, ejecución directa o indirecta si la realiza un humano o un programa (*JUnit* por ejemplo).

- **Caja blanca vs Caja negra.**

  Las pruebas de Caja Blanca comprueban el cómo un sistema se ejecuta, es decir, **se centran en el mecanismo interno** del programa comprobando paso a paso cada una de las acciones del mismo.

Las pruebas de Caja Negra **ignoran el interior** centrándose únicamente en las **entradas y salidas** del sistema.

- **Manuales vs Automáticas.**

  Las pruebas Manuales son **ejecutadas por un usuario**. La fiabilidad siempre está en duda ya que al ser realizadas por usuarios estos siempre pueden cometer errores.

  Las pruebas Automáticas **se implementan y son ejecutadas automáticamente**. Son más fiables aunque no siempre son posibles.

Niveles de Pruebas.

Dependiendo del objetivo de las pruebas en el desarrollo se pueden distinguir seis tipos de pruebas:

1. **Pruebas de Unidad.**

   Las Pruebas de Unidad son **evaluaciones** individuales **de una parte** de un programa (unidad o módulo). Normalmente son una combinación de Caja Blanca y Caja Negra ya que analizan la lógica del módulo usando métodos de caja blanca y completan las pruebas con métodos de caja negra para garantizar que cada módulo cumple con las especificaciones.

2. **Pruebas de Integración.**

   Las Pruebas de Integración son pruebas que **comprueban el funcionamiento entre módulos**.

   Existen dos estrategias o acercamientos, uno incremental y otro no incremental.

3. **Pruebas de Sistema.**

   Evalúan el sistema en su conjunto. El objetivo de este tipo de pruebas es comprobar que se cumplen los requisitos funcionales y las especificaciones técnicas del software. Suele ser habitual que se prueba el sistema en un entorno similar al de producción.

Este tipo de pruebas son aplicables a prototipos.

4. **Pruebas de Carga/Estrés.**

   Este tipo de pruebas evalúa los requisitos no funcionales.

   Las Pruebas de Estrés **comprueban el funcionamiento del sistema bajo condiciones no normales** como pueden ser la ausencia de red, la pérdida de un servidor, etc.

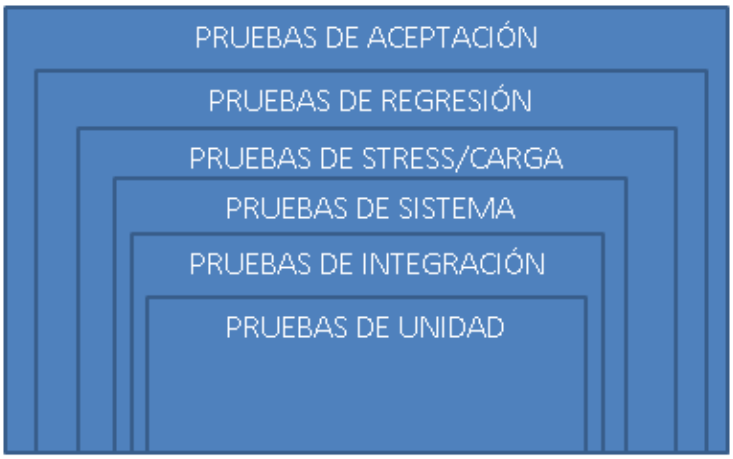
   Las Pruebas de Carga **someten el sistema a cargas de trabajo extremas** determinando la capacidad de resistencia límite del programa, número de usuarios simultáneos, capacidad de cálculo máximo, número de conexiones, etc.

5. **Pruebas de Regresión.**

   Las Pruebas de Regresión sólo son aplicables cuando existen versiones previas del sistema. Consisten en comprobar que las versiones anteriores, o el funcionamiento de las versiones anteriores, siguen estando soportadas por el sistema

6. **Pruebas de Aceptación.**

   Las Pruebas de Aceptación evalúan que el sistema **cumple con los requisitos del cliente**, para ello se cuenta con su participación y por norma general la superación de este tipo de pruebas significa que el cliente acepta el **sistema** y éste queda **validado**.



Por norma general las pruebas de los niveles superiores requieren la superación de las pruebas de los niveles inferiores.

Caja Negra, Caja Blanca.

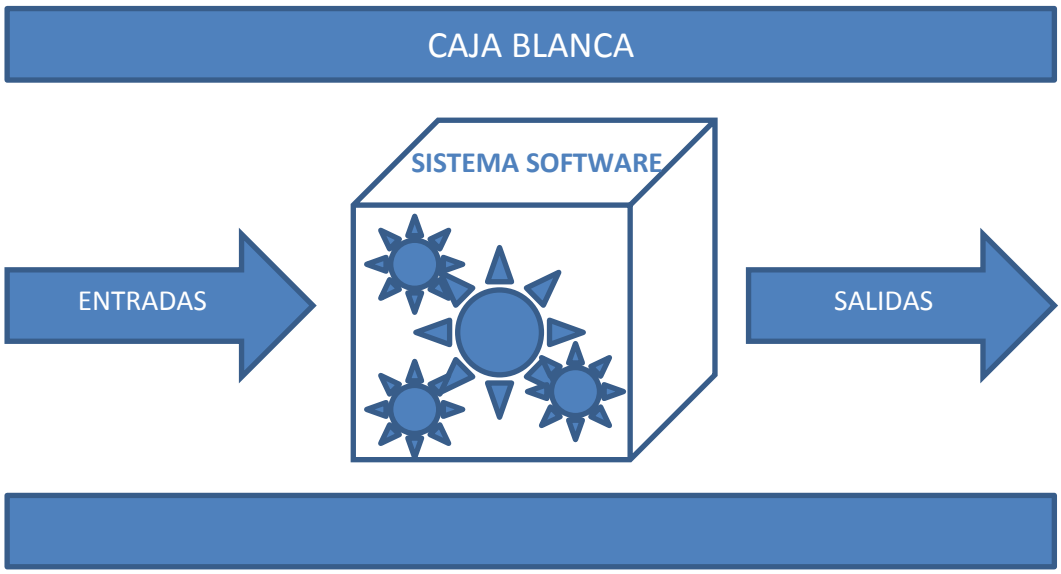
Caja Blanca.

Las pruebas de **Caja Blanca** tienen en cuenta el **funcionamiento interno de un sistema** o componente, es decir, comprueban que el software realiza de forma correcta las diferentes operaciones para las que esté programado.

Aplicar Caja Blanca sobre un sistema facilita la optimización del código ya que se puede observar de forma precisa el funcionamiento a nivel de sentencia de nuestro sistema. Por lo tanto, aumenta la cobertura del mismo.

Otra ventaja de aplicar Caja Blanca es su dinamismo ya que al estar testeando sobre el código este puede alterarse para realizar comprobaciones más exhaustivas.

En contraposición a lo anterior, aplicar Caja Blanca, exige un conocimiento elevado del leguaje de programación y del código del sistema. Este tipo de pruebas dependen directamente de la experiencia del *tester* y en algunas ocasiones es imposible cubrir todos los caminos posibles por lo que puede pasar nuestro sistema, es decir, la cobertura no puede llegar a ser del 100%.



Caja Negra.

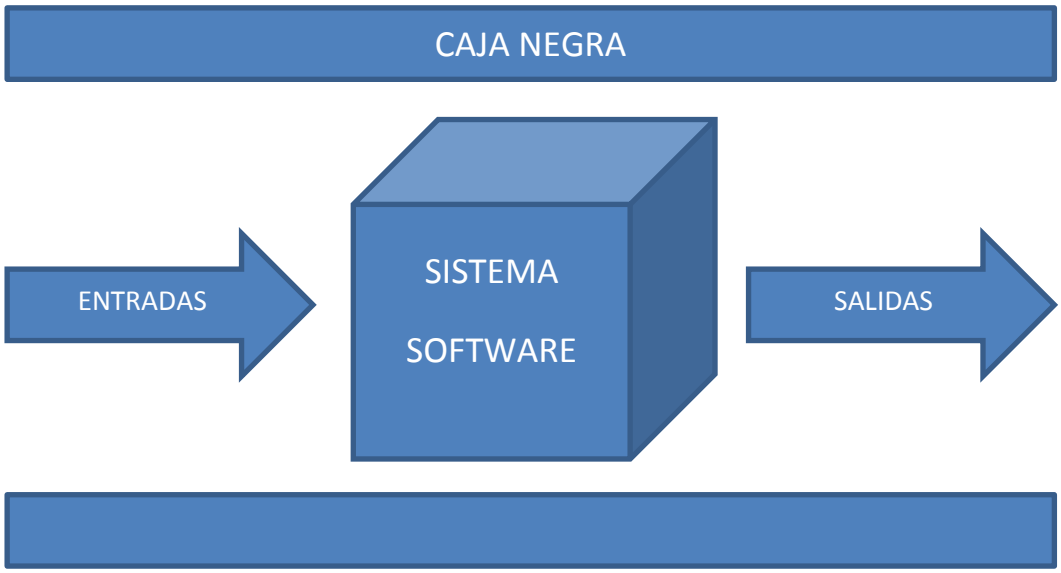
Las pruebas de **Caja Negra** tienen en cuenta únicamente las **entradas y salidas del sistema** o componente, es decir, ignoran el mecanismo interno, el funcionamiento real, a bajo nivel, del software.

Aplicar Caja Negra sobre un sistema posibilita, de forma rápida y sencilla, validar un sistema. Suele ser muy útil cuando nuestro sistema es muy amplio, es decir, contiene grandes cantidades de código. Caja Negra nos informa sobre si un sistema no se comporta según su especificación, es decir, si los resultados para unos datos concretos no son los esperados.

Otra ventaja de Caja Negra es que no se necesita conocer el código porque se realiza sobre las entradas y salidas y por lo tanto pueden ser realizadas por los usuarios del sistema y no sólo por los programadores.

Sus principales deficiencias: el un escaso control sobre la cobertura y la imposibilidad práctica de probar todas las entradas de un sistema.

Para emplear Caja Negra es necesario seleccionar un subconjunto finito de entradas que validen el producto. Esta selección no siempre es sencilla por lo que el nivel de dificultad en el diseño de este tipo de pruebas puede aumentar considerablemente.



No existe un método concreto para determinar que entradas probar. Como norma general se suelen seleccionar los valores límites permitidos y una serie de valores representativos de cada conjunto de entradas.

Por ejemplo para una suma de números enteros positivos se seleccionaría un valor entero positivo grande (4000), un valor positivo normal (10), un valor positivo muy grande (1000000), el valor 0 y un valor negativo para garantizar que el sistema no falla (-3).

El ingenio del *tester* determinará la calidad de las pruebas.

Otra desventaja clara de este tipo de pruebas es que no controlan las dependencias entre entradas, es decir, un sistema puede utilizar las entradas a nivel interno de muchas maneras, generando valores a partir de ellas. Ante infinitas combinaciones de entradas resulta imposible garantizar totalmente que el sistema siempre va a responder correctamente.

Desk checking, *Walkthrough* y Usabilidad.

Algunos tipos de pruebas son realizadas por los programadores durante el desarrollo o la codificación del software. Entre las diferentes técnicas de depuración y factorización de un código fuente destacan: *Desk checking* y *Walkthrough*.

Desk checking.

El ***Desk checking*** es el tipo de prueba más antiguo y quizá uno de los más sencillos. El propio programador busca errores en su código mediante la ejecución manual del mismo.

Este tipo de prueba es imprescindible dentro de la Ingeniería del Software pero suele darse por sentada ya que forma parte de la codificación propiamente dicha, por lo tanto, no cuenta como una prueba principal del sistema, es decir, no doto de calidad al producto ya que se trata de un proceso informal.

Walkthrough.

Walkthrough consiste en simular de forma manual el comportamiento del ordenador, es decir, realizar trazas de ejecución "en papel" para casos de pruebas concretos.

Este tipo de pruebas siempre están orientadas a casos de pruebas sencillos o no demasiado extensos ya que la capacidad para simular el comportamiento de una computadora es limitada.

Lo que se consigue volviendo sobre los pasos, es decir, recorriendo las trazas de ejecución, es obtener información sobre el funcionamiento interno del programa y detectar errores en los caminos lógicos de la aplicación.

Pruebas de usabilidad.

Las **pruebas de usabilidad** consisten en medir las capacidades de uso de un sistema o producto software. Para ello se seleccionan una serie de usuarios que deben probar el sistema en base a tareas predefinidas e incluidas dentro de las funcionalidades del mismo.

Este tipo de pruebas además de comprobar que el sistema contiene y realiza de forma correcta las funcionalidades requeridas permite probar la accesibilidad e intuitividad de los interfaces de usuario.

Pruebas de Unidad.

Podemos definir una **Prueba de Unidad** como un método de **evaluación** de una parte de un sistema, es decir, **de un módulo** de nuestro producto software.

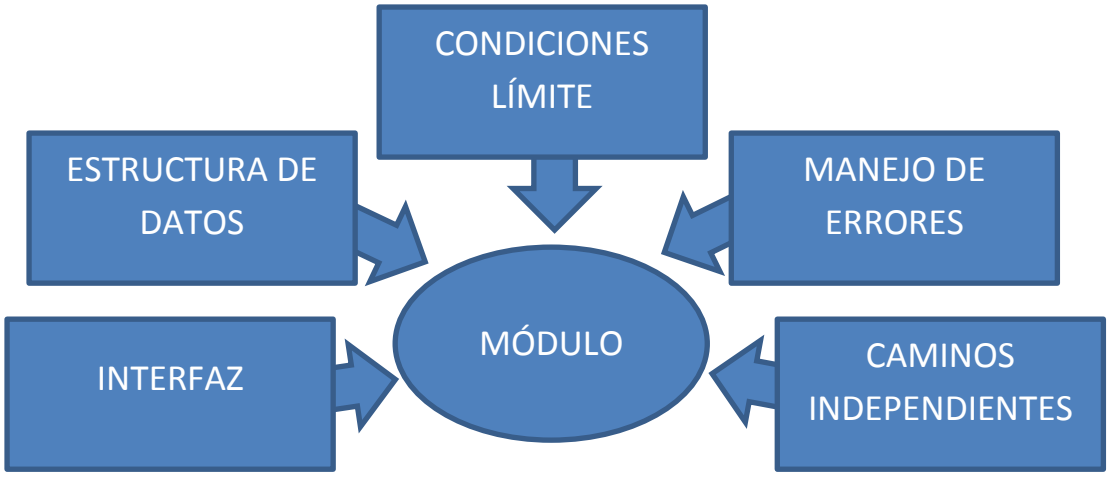
Normalmente las Pruebas de Unidad están constituidas como una mezcla de Caja Negra y Caja Blanca, es decir, combinan la validación de entradas y salidas con la verificación por cobertura del código.

Desde una perspectiva centrada en un módulo concreto podemos definir el proceso de prueba en dos pasos:

1. **Analizar la lógica** del módulo mediante Caja Blanca.
2. **Comprobar** el cumplimiento de los **requerimientos** mediante Caja Negra.

Este tipo de pruebas combina las ventajas tanto de Caja Blanca como de Caja Negra y, al aplicarlas conjuntamente se reducen las desventajas de ambas.

Las Pruebas de Unidad mejoran la documentación del desarrollo, permiten refactorizar el código con confianza y facilitan la corrección de errores o *debugging*.



Pruebas de Integración.

Las **pruebas de integración** se realizan sobre el conjunto del sistema *software*, entendido como la suma de todas sus partes.

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, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.

Las pruebas de integración se pueden subdividir en pruebas incrementales y pruebas no incrementales.

Integración no incremental.

Cuando las pruebas no son incrementales se prueba cada componente por separado y, posteriormente, se integran todos para realizar una prueba final.

Por norma general para evaluar un módulo es necesario simular el resto de módulos con los que tiene alguna dependencia. Posibilita el desarrollo paralelo de módulos ya que éstos son, a efectos prácticos, independientes.

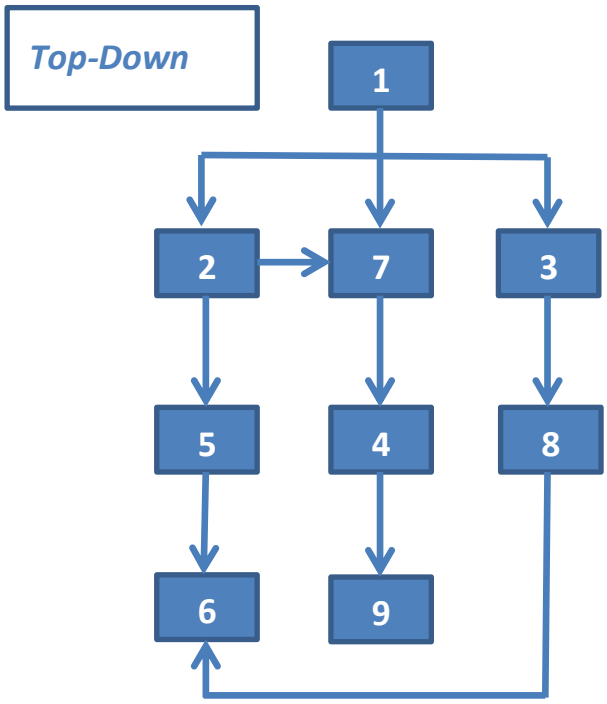
Integración incremental.

Cuando las pruebas son incrementales se combina el siguiente componente que se debe probar con el conjunto de componentes que ya están probados y se va incrementando progresivamente el número de componentes a probar.

Las pruebas de un módulo incluyen las pruebas de todos los anteriores, de esa manera, se garantiza que el funcionamiento de una parte del sistema no afecta a las demás de forma negativa. Existen dos tipos de pruebas de integración incremental: *Top-Down* y *Bottom-Up*.

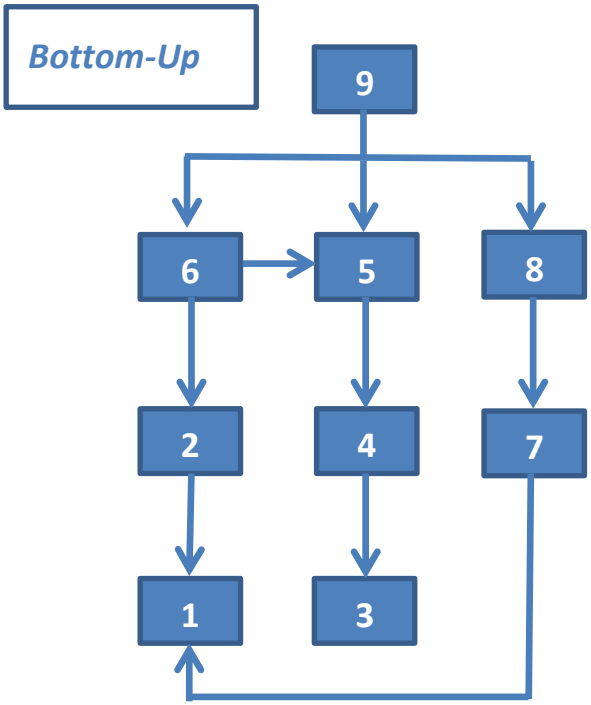
Top-Down.

Top-Down permite generar prototipos en tiempos muy cortos de desarrollo pero suele necesitar un alto nivel de simulación.



Bottom-Up.

Cuando se aplica ***Bottom-Up*** es más sencillo diseñar las pruebas y controlar los resultados de las mismas. Como contrapartida los prototipos se generan mucho más tarde.



En algunas ocasiones un sistema es difícil de probar ya que para hacerlo puede ser necesario que éste interactúe con otros sistemas o con módulos que aún no estén desarrollados.

Normalmente el orden en el que se desarrollan los diferentes módulos suele ser ascendente, es decir, se desarrollan primero las partes a más bajo nivel y se va subiendo en el árbol de dependencias de manera que cuando se desarrolla un módulo concreto todos los módulos que necesita ya han sido desarrollados.

Pero esto no soluciona el problema ya que existen casos donde no se puede desarrollar de forma descendente o donde, al interactuar con sistemas externos, no se tiene acceso a dichos sistemas hasta el final del desarrollo. También puede darse el caso de que el sistema a desarrollar interactúa con un sistema de pago y, por lo tanto, no sea recomendable sobrecargar las pruebas con el mismo ya que se incrementará el coste de producción.

De lo anteriormente expuesto se concluye que debido a la existencia de dependencias, el código no siempre es directamente testeable bien porque existen clases o módulos aún no desarrollados, bien porque se depende de un sistema externo al que no se tiene acceso durante el desarrollo.

Para resolver este tipo de problemas surgen dos herramientas: *Stub* y *Mock*. Son objetos "sustitutos" que simulan el funcionamiento de aquellos elementos no implementados.

Stub.

Los ***Stubs*** implementan la misma lógica de funcionamiento que la parte sustituida pero de una forma más sencilla. No tendría sentido que el tiempo y los recursos necesarios para construir un *Stub* determinado fueran iguales o similares a los necesarios para construir el objeto real, ya que de ser así, sería mejor implementar el real.

Los *Stubs* suelen utilizarse cuando no se tiene acceso a un objeto o sistema externo (servidor web, base de datos de una empresa, etc.), o cuando se quiere reducir el tiempo necesario para realizar las pruebas ya que el tiempo de respuesta de un objeto en memoria siempre será inferior al tiempo de respuesta de un sistema externo.

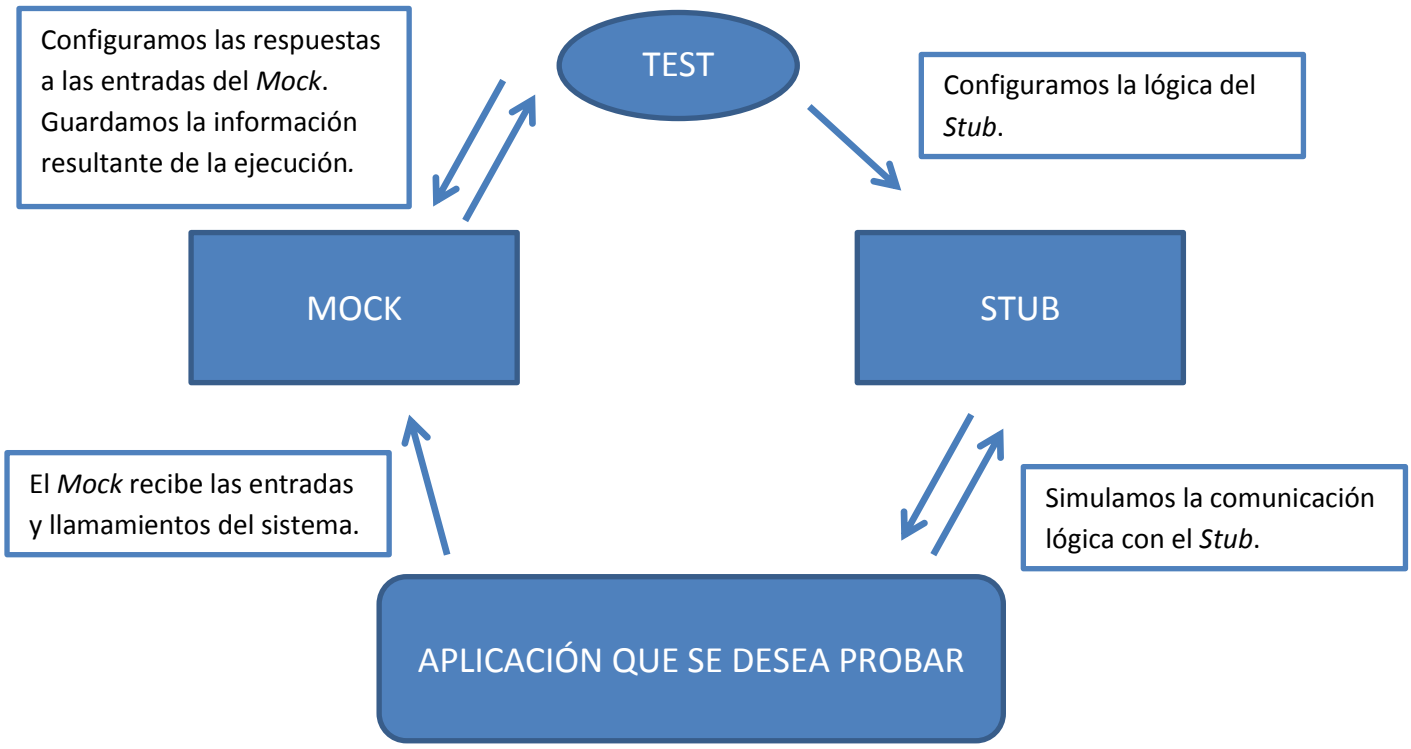
El principal problema de utilizar *Stubs* radica en que implementar la lógica de un sistema u objeto no siempre es sencillo y se corre el riesgo de cometer errores o no ser preciso en dicha simulación. Al mismo tiempo, si el sistema cambia durante el desarrollo mantener los *Stubs* suele ser bastante complicado.

Mock.

Los ***Mocks*** simulan el funcionamiento de la parte sustituida ofreciendo el mismo comportamiento que el esperado pero de forma artificial, es decir, un *Mock* no implementa la lógica del objeto a sustituir, en lugar de ello, lo que hace es simular las salidas esperadas para unas entradas concretas.

Los *Mocks* suelen emplearse cuando es necesario realizar pruebas muy concretas sobre un funcionamiento en particular. Los *Mocks* permiten controlar de forma precisa el comportamiento del sistema ante unas entradas concretas. También suele utilizarse esta técnica cuando implementar la lógica de un *Stub* es demasiado costoso o complicado.

El principal problema de utilizar un *Mock* es que es necesario configurarlo previamente para cada prueba ya que son objetos "estáticos".



“Buenas prácticas” para el *Software Testing*.

Como toda metodología, el *Software Testing* cuenta con una serie de principios que configuran las “buenas prácticas” que se deben emplear cuando se quiera testear un sistema. La utilización de estos principios no garantiza que la tarea de validar y verificar un producto *software* tenga éxito pero aumenta las posibilidades de que lo tenga.

10 Principios para las Pruebas de software.

1. **Una parte necesaria de un acaso de prueba es una definición de la salida o de los resultados esperados.**

   Parece obvio pero si se pasa por encima suele ser una fuente de error más frecuente en los desarrollos comerciales.

   Es importante que toda descripción de un caso de uso esté formada por una descripción de los datos de entrada del programa, una descripción precisa de la salida correcta del programa para el conjunto de datos de entrada concreto.

2. **Un programador debe evitar intentar probar su propio programa.**

   El programador tiende a ver lo que se supone que debe hacer el código en lugar de lo que hace. Cuando un agente externo revisa un sistema suele detectar errores que pasan desapercibidos por su simplicidad a los ojos del programador.
   Si el programador prueba su propio programa se corre el riesgo de no controlar fallos de interpretación sobre la especificación.

   Este principio no se aplica a *debugging* donde, como es evidente, resulta más efectivo que sea realizado por el programador.

3. **Una organización de programación no debería probar sus propios programas.**

   Las organizaciones suelen dar más importancia a los plazos de entrega que a la fiabilidad del software.

   En muchos casos prácticos el proceso de pruebas puede verse como un factor que influye negativamente en los plazos de entrega.

Lo más recomendable dentro de una organización es que existan dos equipos diferenciados, uno de desarrollo y otro de pruebas.

4. **Cualquier proceso de prueba debe incluir una minuciosa inspección de los resultados de cada test.**

   Por norma general los errores que se encuentran en pruebas tardías suelen haber sido omitidos en pruebas previas, es decir, si las pruebas finales de un proyecto detectan errores de relevancia es que las pruebas previas que se han realizado a lo largo de todo el proyecto no han sido correctas.

   Es muy importante valorar los resultados de los test para determinar si realmente el programa cumple con las especificaciones y está correcto.

   Una correcta organización de los test permite una mejor localización de los errores y facilita su corrección.

5. **Deben escribirse casos de prueba para condiciones de entrada que sean inválidas o inesperadas, así como para aquellas que son válidas y esperadas.**

   La tendencia natural es centrar las pruebas en las condiciones válidas de entrada olvidándose de las inválidas. Si no se evalúan los casos no esperados o inválidos es probable que los errores surjan cuando los usuarios tengan acceso al mismo.

6. **Examinar un programa para ver si no hace lo que se supone que debe hacer es sólo la mitad del trabajo, la otra mitad es comprobar que el programa hace lo que se supone que no debe hacer.**

   Puede verso como una parte o derivación del punto anterior. Los programadores pueden no tener clara la función específica del software o sus limitaciones.

7. **Evitar usar y tirar casos de prueba a menos que el programa realmente sea de usar y tirar.**

   Todas las pruebas tienen valor en cuanto a que dotan de valor al producto software. Las pruebas requieren una inversión de tiempo y recursos importantes que deben computarse y, al mismo tiempo, generan documentación sobre el funcionamiento del sistema y sobre sus posibles límites y fallos lo que puede servir como garantía de calidad (valor añadido) o como contrato entre el cliente y el desarrollador.

Si descartamos las pruebas después de usarlas perderemos este valor añadido, por ello se deben guardar y documentar correctamente.

8. **No planificar una prueba asumiendo que no se van a encontrar errores.**

   Es un error común y suele producirse por un exceso de confianza al asumir que una parte o un programa entero no necesita ser probado por ser correcto o no contener errores.

9. **La probabilidad de existencia de más errores en una sección de un programa es proporcional al número de errores ya encontrados en dicha selección.**

   Suele existir una relación lineal entre el número de errores encontrados y el número de errores por descubrir. A efectos prácticos podríamos determinar que los errores trabajan en grupo. Esto es debido a que cuando se produce un error este se propaga por todas las dependencias produciendo otros errores y, al mismo tiempo, cuando una parte concreta de un sistema contiene varios errores puede deducirse que es de baja calidad y por lo tanto es muy probable que contenga más errores.

   Estas afirmaciones permiten centrar los esfuerzos de testeo y corrección sobre aquellas partes que presenten una menor calidad o que sean críticas para el sistema.

10. **Las pruebas de software son una tarea extremadamente creativa y un reto intelectual.**

    Es probable que probar un programa grande o muy complejo sea más complicarlo que diseñarlo. Muchas veces la lógica a implementar para probar un sistema es igual o más compleja que el propio sistema.

Introducción al *Test Driven Development*.

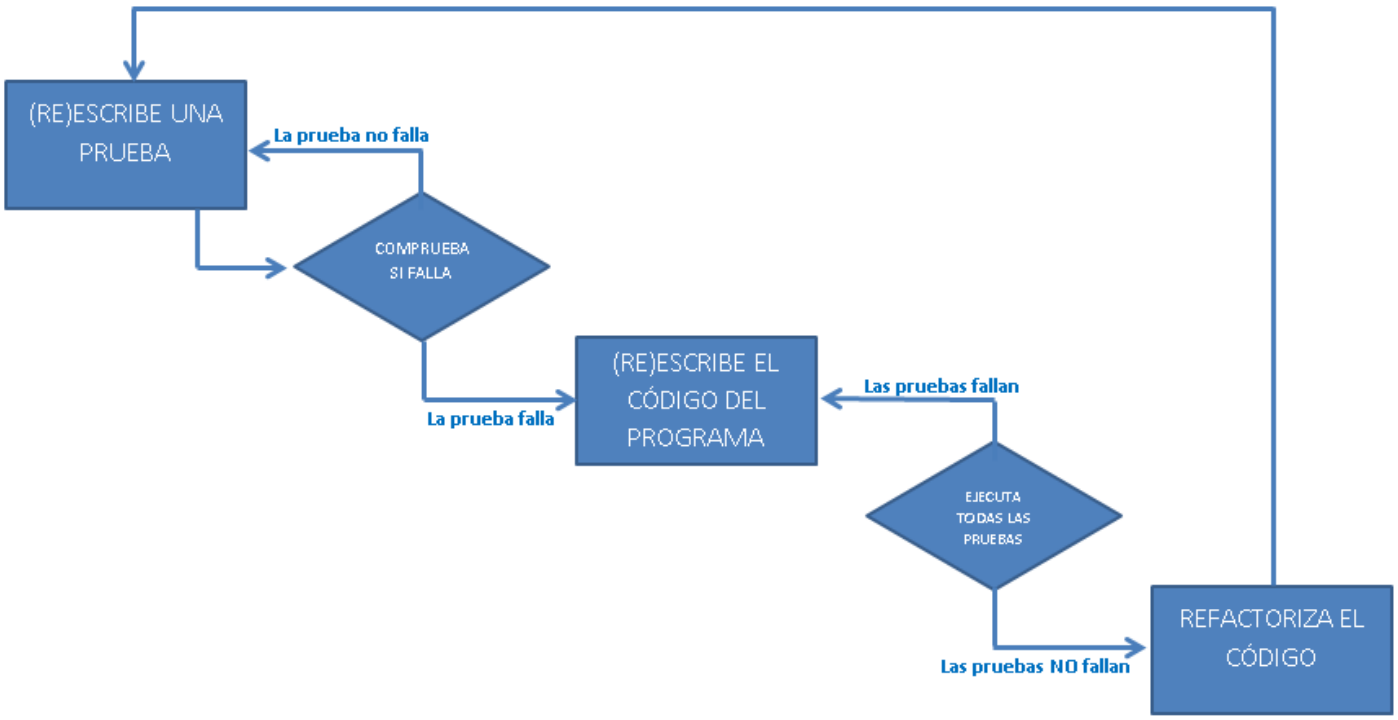
TDD es un modelo de **desarrollo** de software **dirigido por pruebas**, es decir, las pruebas constituyen el motor que va guiando y estructurando todo el proceso de creación de software.

TDD involucra dos prácticas: Escribir las pruebas primero (*Test First Development*) y Refactorización (*Refactoring*).

A grandes rasgos, se escribe una prueba y se verifica que la prueba falla. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito.

El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos en pruebas, de este modo, cuando las pruebas se superen quedará garantizado que el software cumple con los requerimientos que se han establecido previamente.

Funcionamiento del TDD:



La existencia de una metodología orientada por pruebas constituye el último estado de la evolución de las pruebas a lo largo de la historia de la informática. Desde la marginación o la optatividad de las primeras etapas, pasando por la necesidad de tener pruebas al final del desarrollo o al final de cada una de sus

fases; hasta el extremo en el que las pruebas estructuras la metodología por la que se genera el *software*.

Esta evolución de menos a más indica, perfectamente, la importancia que tienen las pruebas dentro de la Ingeniería del *Software* ya que su peso real aumenta de forma paralela al aumento de la complejidad de los sistemas *software*.

Un poco de historia sobre TDD.

En un principio, pensando en el punto de vista más lógico, en un proceso de desarrollo *software* el *testing* se basaba en el concepto de probar después de codificar, *Test after coding* (TAC), es decir, realizar las pruebas (*test*) necesarias para verificar un código que ya había sido escrito con anterioridad. Como curiosidad en los años 60, un proyecto de la NASA, Proyecto *Mercury*, hizo referencia a ciertas formas de desarrollar software mediante el testeo previo a la codificación.

Posteriormente y con la idea de conseguir un desarrollo *software* más práctico, aparece el concepto de programación extrema (*eXtremeProgramming*), dando una mayor importancia a las pruebas.

Se da por supuesto que es imposible prever, y por tanto, cubrir desde el principio todo lo que exigirá la aplicación a desarrollar. Por ello, lo que pretende este tipo de metodología es conseguir una estrecha colaboración con el cliente, implicándole en el proyecto mediante pequeñas versiones de prueba (pruebas unitarias) que le muestran como será la aplicación final.

Estas pruebas unitarias le permiten conocer como será la aplicación y detectar posibles problemas, fallos o necesidades que de otra forma no se verían hasta el final del proceso de desarrollo.

De esta filosofía nacería el *Test-Driven Development* o TDD.

Tres características de TDD.

1. Pruebas unitarias.

Una prueba unitaria, en el mundo de la programación, es cada una de las pruebas que verifican el buen funcionamiento de un módulo de código, así se asegura que cada mini versión de la aplicación final funciona correctamente por separado.

Después, mediante otras pruebas (pruebas de integración) se comprobará el correcto funcionamiento de la aplicación completa.

Las pruebas unitarias son una herramienta de gran importancia tanto para el personal de pruebas como para los desarrolladores. En el caso de la metodología TDD, los *test* unitarios no son los *test* comúnmente desarrollados para probar que las cosas funcionen según lo especificado.

Por ello, los *tests* deben ser automatizados, fáciles de repetir o modificar, y que permitan ser ejecutados continua y consecutivamente de una manera automática con el mínimo esfuerzo posible.

La ejecución de un *test* debe producir un resultado inmediato de éxito, fallo o error.

2. Refactorización.

Martin Fowler, uno de los padres de esta técnica y actualmente uno de los mayores expertos del desarrollo de software ágil y orientado a objetos, dijo:

"Refactorizar es realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo".

Por tanto, el término refactorizar hace referencia a la acción de realizar modificaciones al código fuente asegurando que, tras esta transformación, no se produzca ningún cambio en el comportamiento final del código o de la aplicación, es lo que conocemos informalmente por limpiar el código, ni arregla errores ni añade funcionalidad.

Por lo tanto, lo que se intenta lograr con la refactorización es mejorar la facilidad de comprensión del código, eliminar código muerto o cambiar la estructura de la aplicación, con el fin de facilitar su mantenimiento en un futuro.

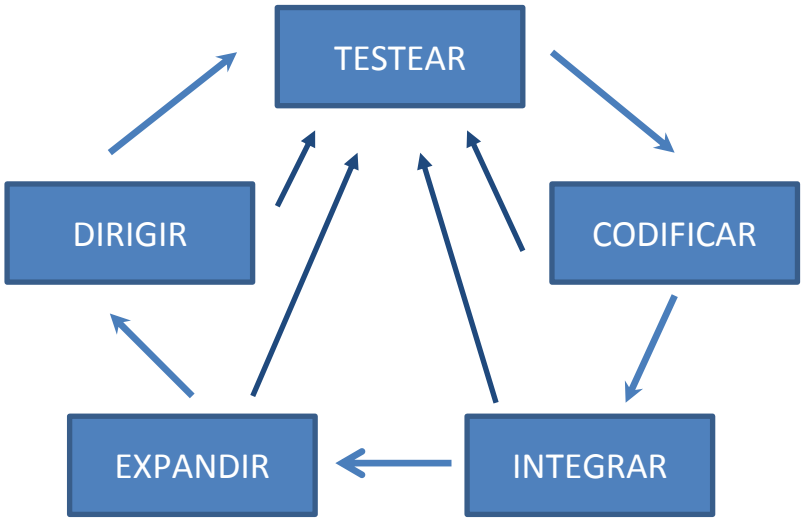
La refactorización se realiza como parte del proceso de desarrollo del software, alternando la inserción de nuevas funcionalidades y casos de prueba de la aplicación, con la refactorización del código para mejorar su consistencia interna y su claridad.

3. Desarrollo iterativo e incremental.

Es una de las prácticas recomendadas en las metodologías agiles como es el caso de la TDD. Está dirigida por las pruebas y la refactorización periódica.

Su objetivo es que la funcionalidad vaya incrementándose en sucesivas iteraciones con sus respectivas revisiones y pruebas en cada una de ellas. Por ello, no se le da a un programa toda su funcionalidad en una iteración, sino que se le va añadiendo y mejorando en cada versión, a medida que transcurran las iteraciones en el proceso de desarrollo el *software* irá creciendo tanto en funcionalidad como en calidad.

El hecho de ir añadiendo módulos hace que el desarrollo sea menos caótico, aspecto importante en toda metodología de desarrollo de *software*.



Entre las ventajas del TDD destacan su flexibilidad y la rapidez de los resultados además de la capacidad de regresión automática para modificar aspectos que otras metodologías resultarían excesivamente costosos.

Los objetivos de TDD.

El objetivo principal del TDD se puede definir de una manera muy simple con la siguiente afirmación, un código limpio que funcione.

1. Código limpio.

Para conseguir un código limpio, dicho código debe cumplir una serie de características como una buena legibilidad, debe estar ausente de duplicaciones, redundancias y mantener un alto nivel de mantenibilidad.

2. Código que obedece a una especificación.

Esta comprobación se realiza mediante el resultado de los *test*, tanto con los *test*. Todo el código de la aplicación se encuentra testeado, documentado y revisado. Las pruebas que se hayan realizado sirven como documentación "contractual" de los límites del dicho código, es decir, garantizan su correcto funcionamiento bajo unas condiciones concretas.

Que la totalidad del código se encuentre verificado y validado asegura tanto un alto nivel de calidad en el producto *software* como el correcto funcionamiento a largo plazo del mismo.

3. Facilitar la resolución de problemas.

Conseguir que los problemas complejos que surgen de la programación, o del diseño de sistemas *software* puedan ser simplificados por los desarrolladores de una forma predecible, evitando, de esta manera, estancamientos en su solución.

4. Flexibilizar el desarrollo.

Este tipo de metodología permite dejar las opciones de desarrollo abiertas para estudiar diferentes caminos o soluciones a un mismo problema. De esta manera se logra conseguir que cualquier cambio no produzca un gran impacto en el desarrollo final del producto *software*.

Glosario.

¿Qué es el software?

Programas de ordenador y la documentación asociada. Los productos de software se pueden desarrollar para algún cliente en particular o para un mercado general.

¿Qué es la ingeniería del software?

La ingeniería del software es una disciplina de ingeniería que comprende todos los aspectos de la producción de software.

¿Qué son las pruebas de software?

Las pruebas de software pueden definirse como una actividad donde un sistema o una parte de un sistema que es ejecutado de forma controlada, es decir, bajo unas condiciones específicas para observar su comportamiento y registrarlo.

¿Qué es un método?

Un método es un enfoque estructurado para el desarrollo de software cuyo propósito es facilitar la producción de software de alta calidad de una forma costeable. Los métodos definen la secuencia de acciones a realizar dentro de una tarea para satisfacer una necesidad concreta.

¿Qué es una tarea?

Una tarea es una acción concreta o la unidad atómica de expresión de todas aquellas acciones que se necesitan para alcanzar un objetivo concreto.

¿Qué es una metodología?

Una metodología es un conjunto de métodos que se aplican de una forma predefinida y constante.

¿Qué es validar un sistema?

Validar un sistema es un proceso por el cual se evalúa una parte o la totalidad de un sistema para determinar si satisface todos los requisitos especificados. Un sistema puede ser validado durante el desarrollo o al final del mismo.

¿Qué características tiene que tener un software de calidad?

Los atributos de un buen software son: mantenibilidad, confiabilidad, eficiencia y usabilidad.

¿Qué es mantenibilidad?

El software debe escribirse de tal forma que pueda evolucionar para cumplir las necesidades de cambio de los clientes. Éste es un atributo crítico debido a que el cambio en el software es una consecuencia inevitable de un cambio en el entorno de negocio.

¿Qué es confiabilidad?

La confiabilidad del software tiene un gran número de características, incluyendo la fiabilidad, la protección y la seguridad. El software confiable no debe causar daños físicos en el caso de que se produzca un error en el sistema.

¿Qué es eficiencia?

El software no debe hacer que se malgasten los recursos del sistema, como la memoria y los ciclos de procesamiento. Por lo tanto, la eficiencia incluye tiempos de respuesta y de procesamiento, de utilización de memoria, etc.

¿Qué es usabilidad?

El software debe ser fácil de utilizar, sin esfuerzo adicional, por el usuario para quien está diseñado. Esto significa que debe tener una interfaz de usuario apropiada y una documentación adecuada.

¿Qué es verificar un sistema?

Verificar un sistema es un proceso por el cual se determina si los productos de una fase de desarrollo determinada satisfacen las condiciones impuestas al principio de esa fase. De forma informal podría definirse como la comprobación de si “el código” hace lo que tiene que hacer.

¿Qué es el *checkout*?

Se define como *checkout* al conjunto de pruebas realizadas en el entorno operacional (soporte). Dichas pruebas deben garantizar que el producto software funciona como se requiere después de la instalación.

¿Qué es el *debug*?

Se define como *debug* a la tarea de detectar, localizar y corregir fallos en un programa dado. Incluye el uso de técnicas de depuración como puntos de ruptura, *desk checking*, volcados de memoria, inspección de código fuente, uso de *flags*, trazas, etc.

¿Qué es un proceso del software?

Un conjunto de actividades cuya meta es el desarrollo o evolución del software.

¿Qué son los métodos de la ingeniería del software?

Enfoques estructurados para el desarrollo de software que incluyen modelos de sistemas, notaciones, reglas, sugerencias de diseño y guías de procesos.

¿Cuáles son los atributos de un buen software?

El software debe tener la funcionalidad y el rendimiento requeridos por el usuario, además de ser mantenible, confiable y fácil de utilizar.

¿Qué es un *tester*?

Un *tester* o probador es una persona que se encarga de realizar las pruebas de un sistema. Por norma general suelen poseer buenas habilidades de desarrollo y codificación, conocimientos de algoritmos y leguajes formales.

¿Qué es cobertura?

La cobertura, cuando hablamos de casos de prueba, hace referencia al grado en el que las pruebas cubren la lógica del sistema, es decir, todos los caminos posibles que puedan trazarse sobre las sentencias del código, o lo que es lo mismo, todas las posibles combinaciones que se puedan producir durante su ejecución.

¿Qué es un *test* de unidad?

Un *test* de unidad es una prueba concreta que se realiza sobre un módulo o "unidad" mínima autosuficiente dentro de un sistema. Dentro del paradigma orientado a objetos, por norma general un *test* de unidad prueba el funcionamiento una clase concreta.

¿Qué es un interfaz de usuario?

Un interfaz de usuario es un medio con el que el usuario puede comunicarse con una máquina. Por norma general suelen ser fáciles de entender y utilizar.

¿Qué es un escenario?

Los escenarios describen situaciones del sistema, es decir, enuncian las condiciones o requisitos del sistema.

¿Qué es el entorno de software?

El entorno de software contiene toda la información sobre la configuración de un sistema (módulos, servicios de red, controladores, informes de errores, etc).

¿Qué es el refactorizar código?

El término refactorización se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, es decir, informalmente se limpia el código. De esta manera el código fuente resultante suele ser más sencillo, estar mejor estructurado y ser más legible.

Bibliografía.

- Kent Beck, *Test-driven development: by example*, 1ª. Ed., 2003.
- Whittaker, J.A. *What Is Software Testing? And Why Is It So Hard?* IEEE Software, 17(1).
- Tassey, G. *The Economic Impact of Inadequate Infraestructure for Software Testing*. Informe del RTI para el NIST.
- Myers, G.J. *The Art of Software Testing*. John Wiley & Sons, Inc.
- *Software Testing Techniques Technology Maturation and Research Strategies.* Lu Luo. School of Computer Science. Carnegie Mellon University.
- *P. Santhanam y B. Hailpern, Software debugging, testing, and verification*.
- *Diseño Ágil con TDD*. Autores: Carlos Blé Jurado, Juan Gutiérrez Plaza, Fran Reyes Perdomo y Gregorio Mena.