



# Pruebas de software: Técnicas

---

Jhonny Alexander Parra  
Miguel Angel Chaves

# Agenda

Pruebas de Software: Técnicas

Introducción

Objetivos

Error, defecto, fallo

Etapas de prueba

Tipo de pruebas: *Unitarias*,  
*integración*, *alto nivel*

Caja blanca y negra

Test Driven Development

Herramientas

---

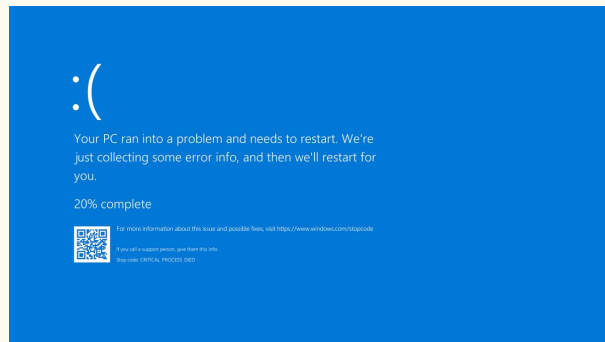
# Introducción: ¿Para qué probar?

Video de introducción:

<https://www.youtube.com/watch?v=AbER5TVTHEc>

Sí, el software falla.

No todas las fallas de software son para reírse.



# ¿Para qué probar el software?



26 de abril de 1994 en Japón.

Un accidente de avión  
provoca la muerte de 264  
personas.

¿Error del piloto o del  
software?

(<https://www.youtube.com/watch?v=85zPfq7etI>)

# ¿Para qué probar el software?

1985 en Canadá.



La Therac-25, una máquina de radioterapia suministró dosis letales de radiación causando la muerte de 3 pacientes por causa directa. Todo por un bug en el software que controlaba la máquina.

# ¿Para qué probar el software?



4 de junio de 1996.

Un bug de software en el lanzamiento del Ariane 5.

500 millones de dólares perdidos y meses de desarrollo desaprovechados.

(<https://www.youtube.com/watch?v=5tJPXYA0Nec>)

# Introducción: ¿Para qué probar el software?

- El software está involucrado en entornos que no admiten fallos: bancos, hospitales, aviones, el mundo de los negocios, etc.
- Probar software es importante porque las fallas pueden ser sumamente costosas y peligrosas.
- Las fallas en el software pueden traer consecuencias catastróficas como pérdidas humanas o pérdidas de grandes sumas de dinero.
- Incluso en softwares pequeños las pruebas no son opcionales.



# Objetivos de probar el software

Las pruebas intentan demostrar que un programa hace lo que se intenta que haga, así como descubrir defectos en el programa antes de usarlo.

Demostrar al desarrollador y al cliente que el software cumple con los requisitos. Una prueba por cada requisito o característica.



Encontrar situaciones donde el comportamiento del software es incorrecto. Esto para erradicar los comportamientos indeseables, como caídas del sistema, interacciones no deseadas con otros sistemas, cálculos incorrectos y corrupción de datos.





# Error, defecto y fallo

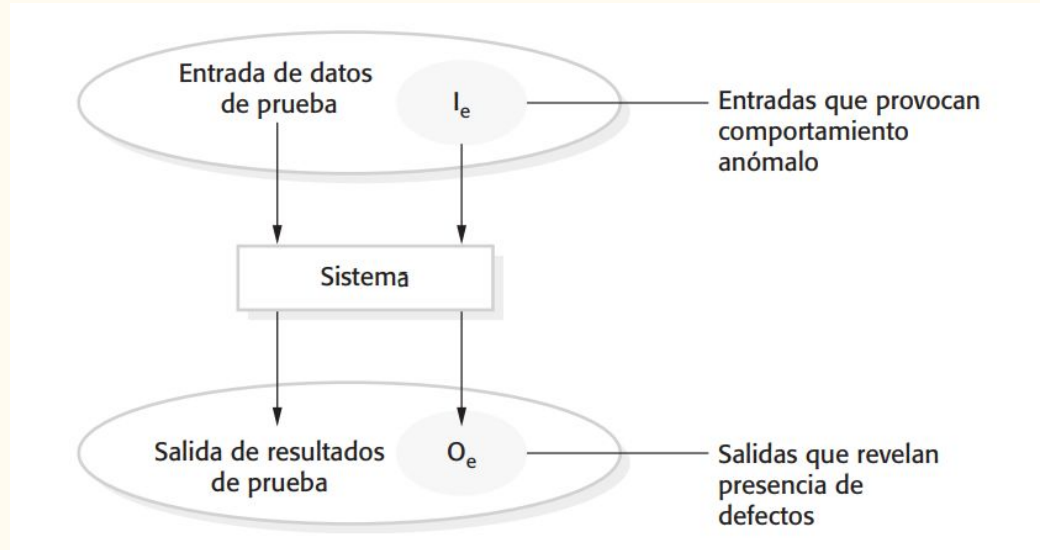
**Error:** Una equivocación de una persona al desarrollar una actividad de desarrollo de software.

**Defecto:** Desperfecto en un componente que puede causar que el mismo falle en sus funciones.

**Fallo:** Manifestación visible de un defecto.

*“Defecto es una vista interna, lo ven los desarrolladores, falla es la vista externa, la ven los usuarios”. [it-mentor]*

# Pruebas de validación y de defecto



Tomada de Ingeniería de Software, Sommerville (2011) [2]

*Las pruebas pueden mostrar sólo la presencia de errores, más no su ausencia.*  
(Dijkstra, 1972)

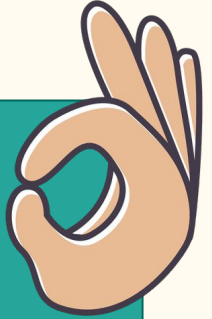
# Validación y verificación

Validación:



*¿Construimos el  
producto correcto?*

Verificación:




*¿Construimos bien el  
producto?*

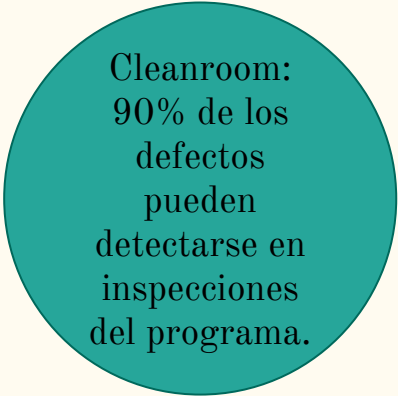
Las pruebas son solo una parte del  
proceso de validación y verificación  
del software.

# Inspecciones o revisiones de software

1. No hay que preocuparse por la interacción entre errores.
2. Las versiones incompletas de un sistema se pueden inspeccionar sin costos adicionales.
3. Una inspección puede considerar atributos más amplios de calidad de un programa, como el cumplimiento de estándares, la portabilidad y la mantenibilidad.



60% de los  
errores se  
encuentran en  
inspecciones  
informales.



Cleanroom:  
90% de los  
defectos  
pueden  
detectarse en  
inspecciones  
del programa.

# ¿Hasta qué punto validar y verificar?

La meta es establecer confianza de que el sistema de software es “**adecuado**”:

## *Propósito de software:*

Cuanto más crítico sea el software, más importante debe ser su confiabilidad.

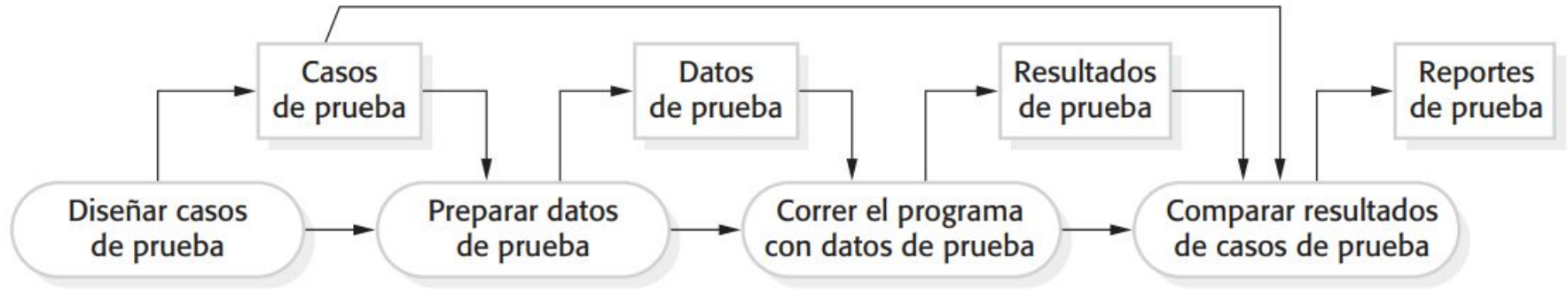
## *Expectativas del usuario:*

Debido a su experiencia con software no confiable y plagado de errores, muchos usuarios tienen pocas expectativas de la calidad de software.

## *Entorno del mercado:*

Si un producto de software es muy económico, los usuarios tal vez toleren un nivel menor de fiabilidad.

# Qué es una prueba: el proceso de prueba tradicional



Modelo del proceso de pruebas de software.

Tomada de Ingeniería de Software, Sommerville (2011) [2]

# Etapas de prueba

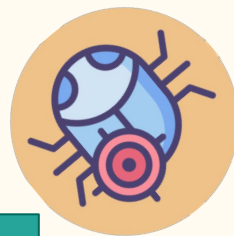
1. **Pruebas de desarrollo**, donde el sistema se pone a prueba durante el proceso para descubrir errores (bugs) y defectos.
2. **Versiones de prueba**, donde un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarla a los usuarios.
3. **Pruebas de usuario**, donde los usuarios reales o potenciales de un sistema prueban el sistema en su propio entorno.

¡Es necesario combinar pruebas manuales y automatizadas!

# Pruebas de desarrollo

Actividades de prueba realizadas por:

- Programador que diseñó el software.
- Parejas programador/examinador.
- Grupo de pruebas independiente al grupo de desarrollo.
- Son ante todo un proceso de prueba de defecto, la meta es descubrir bugs en el software.
- Están relacionadas con el proceso de depuración: localizar problemas en el código y corregirlos.



Pruebas unitarias

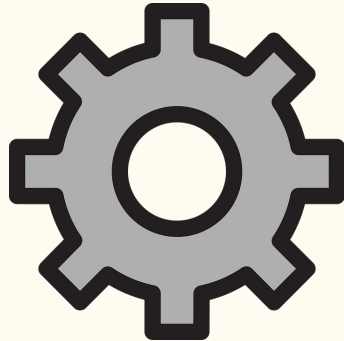
Pruebas de  
componentes

Pruebas del  
sistema



# Pruebas de desarrollo: Pruebas unitarias

- Probar componentes unitarios del programa.
- Las funciones o métodos individuales son el tipo más simple de componente de un sistema.
- POO: Clases. Otros paradigmas: funciones.
- Establecer y verificar el valor de todos los atributos del componente unitario.
- Poner el componente unitario en todos los estados posibles (simular eventos).



*¿Que probaría de un engranaje?*

# Pruebas unitarias: Ejemplo

Probar configuración de atributos.

identificador ¿se configuró bien? ¿es válido?

Pruebas para todos los métodos asociados con la clase: reportWeather(), reportStatus(), etc

Se deben automatizar las pruebas unitarias:

1. Se inicializa el sistema con las entradas y salidas esperadas.
2. Se llama al método que se va a probar.
3. Se compara el resultado obtenido con el resultado esperado.

EstaciónMeteorológica
identificador
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

Tomada de Ingeniería de Software,  
Sommerville (2011) [2]

# Pruebas unitarias: los mocks

**Problema:** En ocasiones, el componente unitario que se prueba tiene dependencias de otros componentes unitarios que no se escribieron o que, si se utilizan, frenan el proceso de pruebas.

**Solución:** Objetos mock (simulados)

- Objetos con la misma interfaz que los objetos usados que simulan su funcionalidad.
- Pueden usarse también para simular una operación anormal o eventos extraños.

# Mock: Ejemplos

- Una base de datos puede requerir de un proceso de configuración lento antes de usarse.
- Un mock que aparenta una base de datos suele tener algunos ítems de datos que se organizan en un arreglo.
- Es más rápido, no hay sobrecargas de llamar a una base de datos o discos.



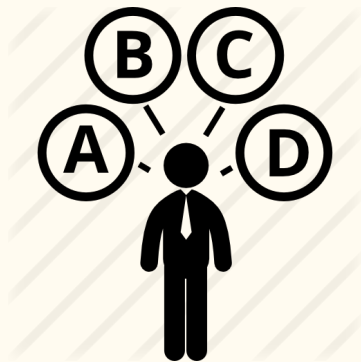
Si se pretende que el sistema tome acciones a ciertas horas del día, el objeto mock simplemente regresará esas horas, independientemente de la hora real del reloj.



# ¿Cómo elegir los casos de pruebas unitarias?

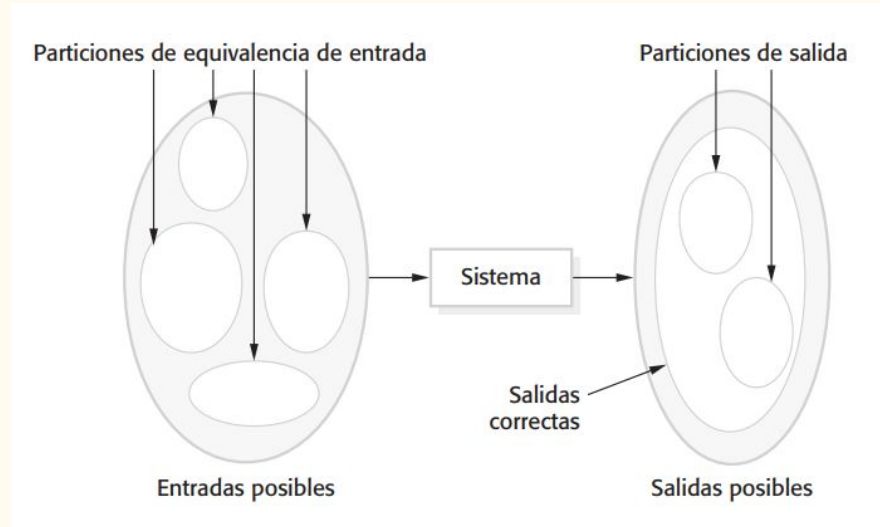
Hacer pruebas es costoso y lleva tiempo.

1. Los casos de prueba tienen que mostrar, que el componente unitario (clase/función) hace lo que se supone que debe hacer cuando se somete a prueba.
2. Si hay defectos en la clase/función, éstos deberían revelarse mediante los casos de prueba.



# Estrategias: Prueba de partición

Se identifican grupos de entradas con características comunes y se procesan de la misma forma. Se deben elegir las pruebas dentro de cada uno de los grupos.



Tomada de Ingeniería de Software, Sommerville (2011) [2]

# Estrategias: Pruebas basadas en lineamientos

Se usan lineamientos para elegir los casos de prueba, dichos lineamientos expresan la experiencia previa de los tipos de errores que suelen cometer los programadores.

Algunos lineamientos:

1. Elegir entradas que fuercen al sistema a generar todos los mensajes de error.
2. Diseñar entradas que produzcan que los buffers de entrada se desborden.
3. Repetir varias veces la misma entrada o serie de entradas.
4. Forzar la generación de salidas inválidas.
5. Forzar resultados de cálculo demasiado largos o demasiado pequeños.



# Pruebas de desarrollo: Pruebas de componentes

- Los componentes de software están constituidos por varios objetos individuales en interacción.
- Hay errores en los componentes que no se detectan al poner a prueba los objetos individuales, porque dichos errores resultan de interacciones entre los objetos en el componente.
- ¿La interfaz del componente unitario se comporta de acuerdo a su especificación?



*¿Qué probarían de un componente con tres engranajes?*



# Pruebas de componentes: errores en interfaces

**Uso incorrecto de interfaz:** Un componente unitario que llama a otro componente unitario comete algún error en el uso de la interfaz. *Ejemplo: Parámetros de tipo equivocado, orden equivocado o número equivocado.*

**Mala interpretación de interfaz:** Un componente unitario malinterpreta la especificación de la interfaz del componente unitario llamado y hace suposiciones sobre su comportamiento. *Ejemplo: se llama un método que hace una búsqueda binaria con un arreglo desordenado.*

**Errores de temporización:** Ocurren en sistemas de tiempo real que usan una memoria compartida o una interfaz que pasa mensajes. El productor de datos y el consumidor de datos operan a diferentes niveles de rapidez.

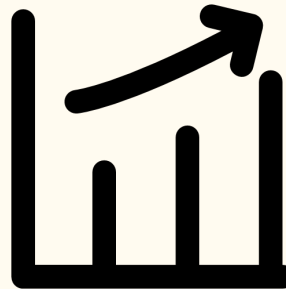
# Lineamientos para las pruebas de interfaz

- Examinar el código que se va a probar y listar explícitamente cada llamado a un componente externo.
- Probar con valores nulos.
- Hacer pruebas que deliberadamente hagan que fallen los componentes unitarios.
- Usar pruebas de esfuerzo en los sistemas que pasan mensajes, es decir, pasar más mensajes de los que probablemente ocurran en la práctica.
- Variar el orden en el que se activan los componentes.



# Integración y pruebas incrementales

- Implican integrar diferentes componentes y, después, probar el sistema integrado.
- Siempre hay que usar el enfoque incremental: integrar un componente y probar, integrar otro componente y probar de nuevo.
- De esta manera se saben que si hay problemas, quizá se deban a las interacciones con el componente que se integró más recientemente.



# Pruebas de desarrollo: Pruebas del sistema

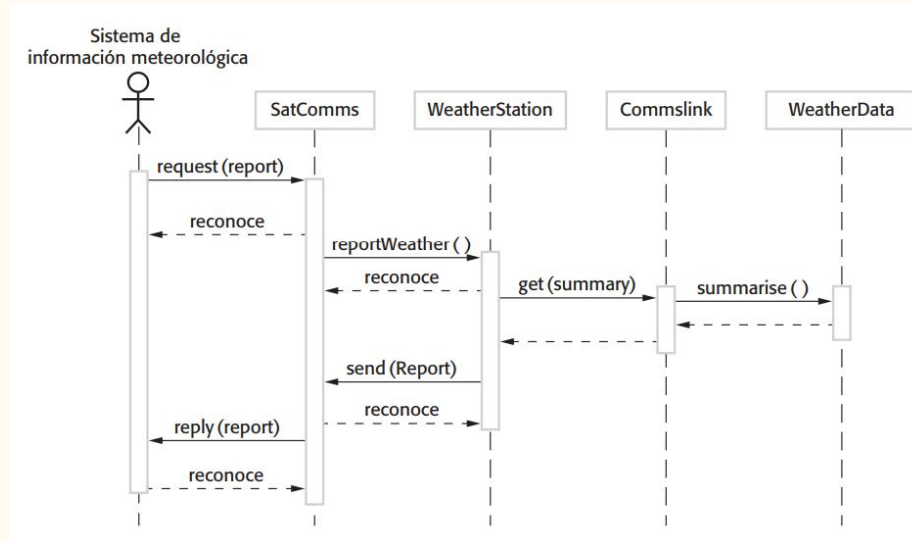
Integración de componentes para crear una versión del sistema y poner a prueba el sistema integrado.

¿Los componentes son compatibles e interactúan correctamente a través de sus interfaces?

1. Los componentes reutilizables desarrollados por separado y los sistemas comerciales pueden integrarse con componentes desarrollados recientemente.
2. Los componentes desarrollados por diferentes miembros del equipo o de grupos pueden integrarse en esta etapa.

# Pruebas de sistema: Casos de uso

Los casos de uso son implementados por varios componentes u objetos del sistema.  
Probar los casos de uso obliga a que ocurran las interacciones entre los componentes.



Recolección de  
información  
meteorológica.

# Pruebas de sistema

Es imposible probar todas las secuencias posibles de ejecución del programa.

Las pruebas se deben basar en un subconjunto de probables casos de prueba:

1. Tienen que probarse todas las funciones del sistema que se ingresen a través de un menú.
2. Debe experimentarse la combinación de funciones que se ingrese por medio del mismo menú.
3. Donde se proporcione entrada del usuario hay que probar todas las funciones con entrada correcta o incorrecta.

# Test Driven Development

# Desarrollo Dirigido por Pruebas

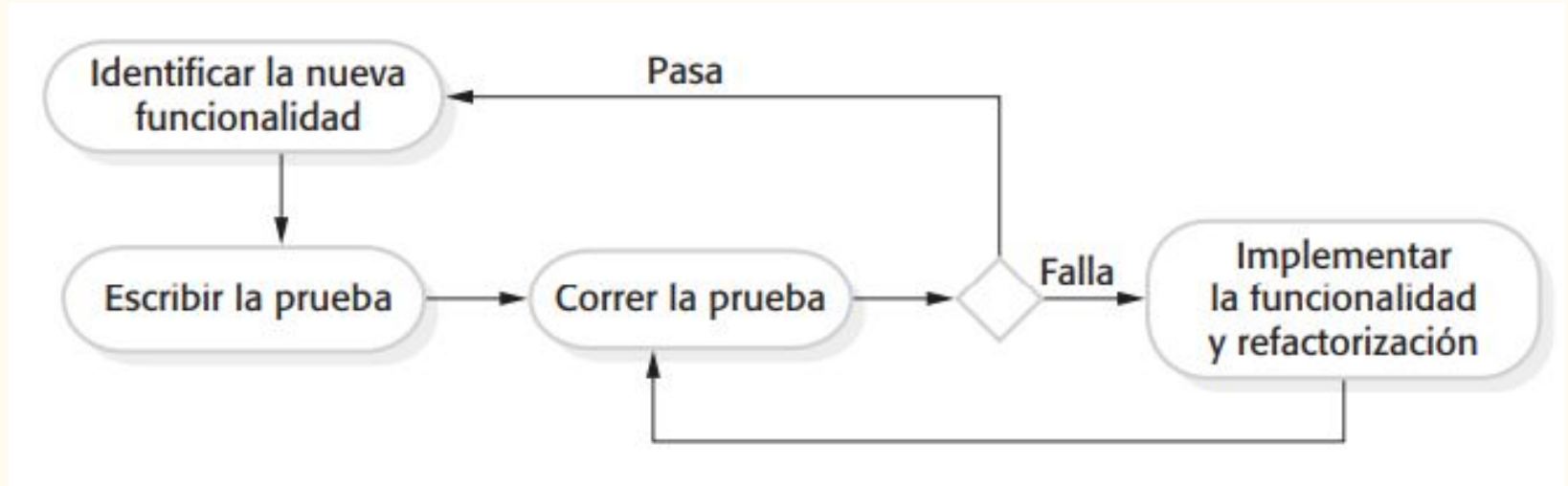
El desarrollo dirigido por pruebas (TDD, por las siglas de Test-Driven Development) es un enfoque al diseño de programas donde se entrelazan el desarrollo de pruebas y el de código (Beck, 2002; Jeffries y Melnik, 2007).

En esencia, el código se desarrolla incrementalmente, junto con una prueba para ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado pasa la prueba.





# Proceso TDD Fundamental



Tomada de Ingeniería de Software, Sommerville (2011) [2]

# ¿Porqué surgió la TDD?

Un argumento consistente con el desarrollo dirigido por pruebas es que ayuda a los programadores a aclarar sus ideas acerca de lo que realmente debe hacer un segmento de código. Para escribir una prueba, es preciso entender lo que se quiere, pues esta comprensión facilita la escritura del código requerido.

***Por ejemplo,** si su cálculo implica división, debería comprobar que no divide los números entre cero. En caso de que olvide escribir una prueba para esto, en el programa nunca se incluirá el código a comprobar.*

# Beneficios

**Cobertura de código:** En principio, cualquier segmento de código que escriba debe tener al menos una prueba asociada.

**Pruebas de regresión:** Un conjunto de pruebas se desarrolla incrementalmente conforme se desarrolla un programa. Siempre es posible correr pruebas de regresión para demostrar que los cambios al programa no introdujeron nuevos bugs.

**Depuración simplificada:** Cuando falla una prueba, debe ser evidente dónde yace el problema. Es preciso comprobar y modificar el código recién escrito.

**Documentación del sistema:** Las pruebas en sí actúan como una forma de documentación que describen lo que debe hacer el código. Leer las pruebas suele facilitar la comprensión del código.

# Pruebas de versión

Ponen a prueba una versión particular de un sistema que pretende estar fuera del equipo de desarrollo.

Diferencias entre las pruebas de versión y las pruebas de sistema:

1. Un equipo independiente que no intervino en el desarrollo del sistema debe ser el responsable de las pruebas de versión.
2. Las pruebas por parte del equipo de desarrollo deben enfocarse en el descubrimiento de bugs (defectos). El objetivo de las pruebas de versión es comprobar que el sistema cumpla con los requerimientos y sea suficientemente bueno (validación).

# Pruebas de versión

Convencer al proveedor de que el sistema es lo suficientemente apto para su uso:

- El sistema entrega la funcionalidad, rendimiento y confiabilidad especificados.
- El programa no falla durante su uso normal.
- Son pruebas de caja negra.
- Es una prueba funcional, al examinador no le preocupa la aplicación del software.

# Pruebas basadas en requerimientos

Son pruebas de validación más que de defecto: se intenta demostrar que el sistema implementa adecuadamente sus requerimientos.

*“Los requerimientos tienen que escribirse de forma que pueda diseñarse una prueba para dicho requerimiento”*

# Pruebas de escenario

Son un enfoque a las pruebas de versión donde se crean escenarios típicos de uso y se les utiliza en el desarrollo de casos de prueba para el sistema.

Un escenario es una historia que describe una forma en que puede usarse el sistema. Los escenarios deben ser realistas, y los usuarios reales del sistema tienen que relacionarse con ellos.

*Kaner (2003) sugiere que una prueba de escenario debe ser una historia narrativa que sea creíble y bastante compleja.*

# Pruebas de rendimiento

Las pruebas de rendimiento deben diseñarse para garantizar que el sistema procese su carga pretendida. Generalmente, esto implica efectuar una serie de pruebas donde se aumenta la carga, hasta que el rendimiento del sistema se vuelve inaceptable.

Para probar si los requerimientos de rendimiento se logran, quizá se deba construir un *perfil operativo*. Un perfil operativo es un conjunto de pruebas que reflejan la mezcla real de trabajo que manejará el sistema.





En las pruebas de rendimiento, significa estresar el sistema al hacer demandas que estén fuera de los límites de diseño del software.

Este tipo de pruebas tiene dos funciones:

- 1. Prueba el comportamiento de falla del sistema.**
- 2. Fuerza al sistema y puede hacer que salgan a la luz defectos que no se descubrirán normalmente.**

# Pruebas de usuario

Los clientes o usuarios proporcionan entrada y asesoría sobre las pruebas del sistema.

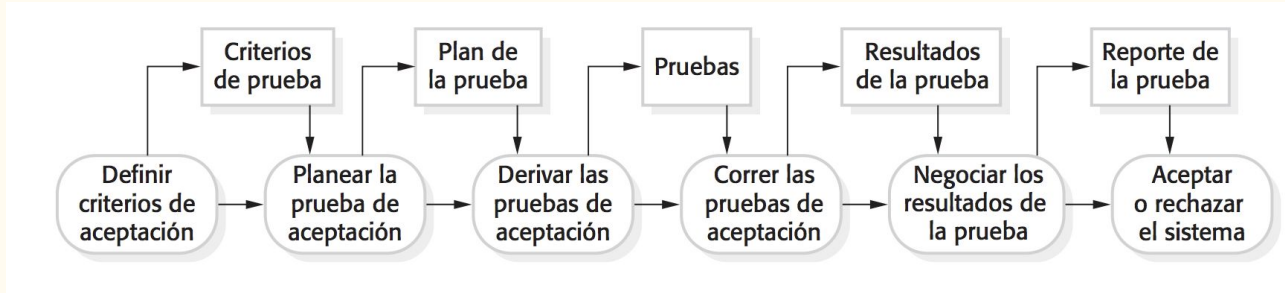
Los desarrolladores no pueden replicar el entorno de trabajo del sistema, las pruebas en el entorno del desarrollador son artificiales.



# Tipos de pruebas de usuario

1. Pruebas alfa, donde los usuarios del software trabajan con el equipo de diseño para probar el software en el sitio del desarrollador.
2. Pruebas beta, donde una versión del software se pone a disposición de los usuarios, para permitirles experimentar y descubrir problemas que encuentran con los desarrolladores del sistema.
3. Pruebas de aceptación, donde los clientes prueban un sistema para decidir si está o no listo para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente.

# Pruebas de Aceptación



Tomada de Ingeniería de Software, Sommerville (2011) [2]

Las pruebas de aceptación son una parte inherente del desarrollo de sistemas personalizados. Tienen lugar después de las pruebas de versión. Implican a un cliente que prueba de manera formal un sistema, para decidir si debe o no aceptarlo del desarrollador del sistema.

Caja blanca y  
negra

# Caja Blanca

Pruebas con acceso al código fuente (datos y lógica). Se trabaja con entradas, salidas y el conocimiento interno.

# Caja Negra

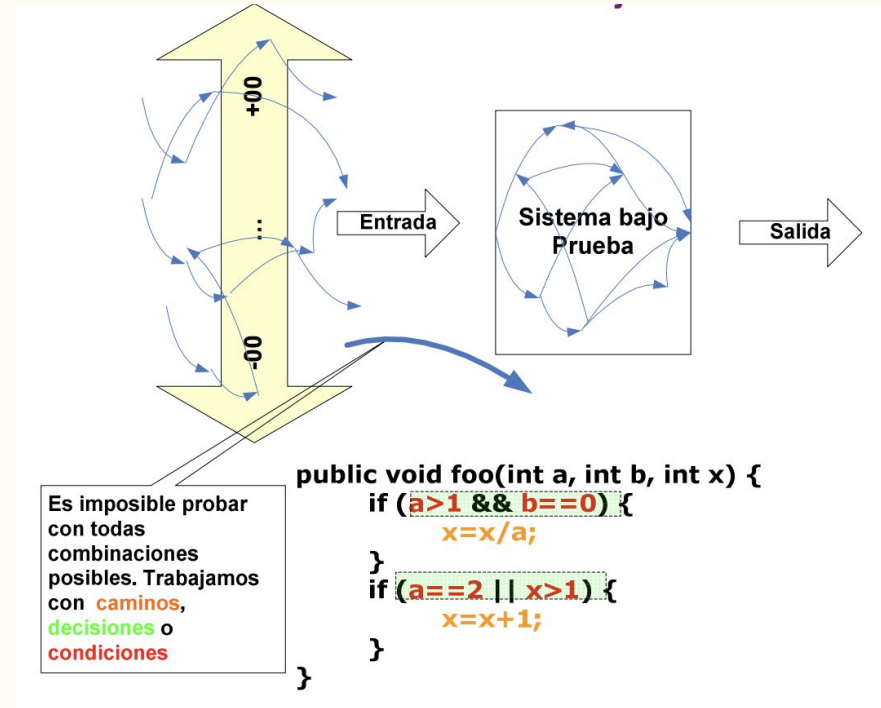
Pruebas funcionales sin acceso al código fuente de la aplicación, se trabaja con entradas y salidas.

---

# Caja blanca

Las pruebas de caja blanca intentan garantizar que:

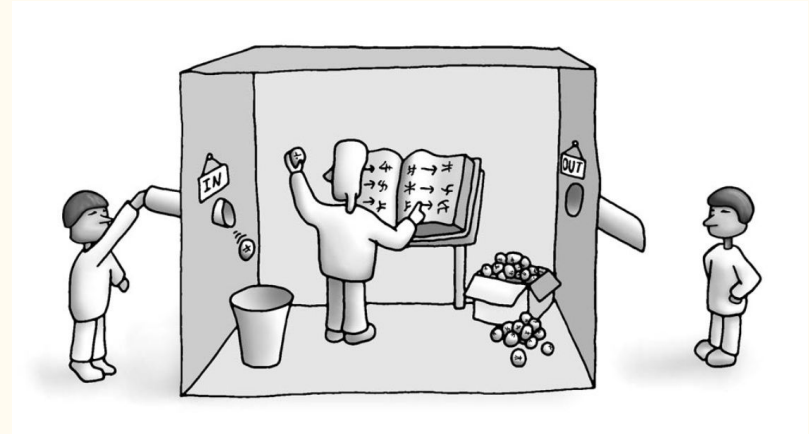
- Se ejecutan al menos una vez todos los caminos independientes de cada módulo
- Se ejecutan las decisiones en su parte verdadera y en su parte falsa
- Se ejecuten todos los bucles en sus límites
- Se utilizan todas las estructuras de datos internas para comprobar su validez.



# Tipo de pruebas de caja blanca

Las principales técnicas de diseño de pruebas de caja blanca son:

- Prueba de la Ruta Básica
- Pruebas de la estructura de control
- Prueba de condición
- Prueba del flujo de datos
- Prueba de bucles

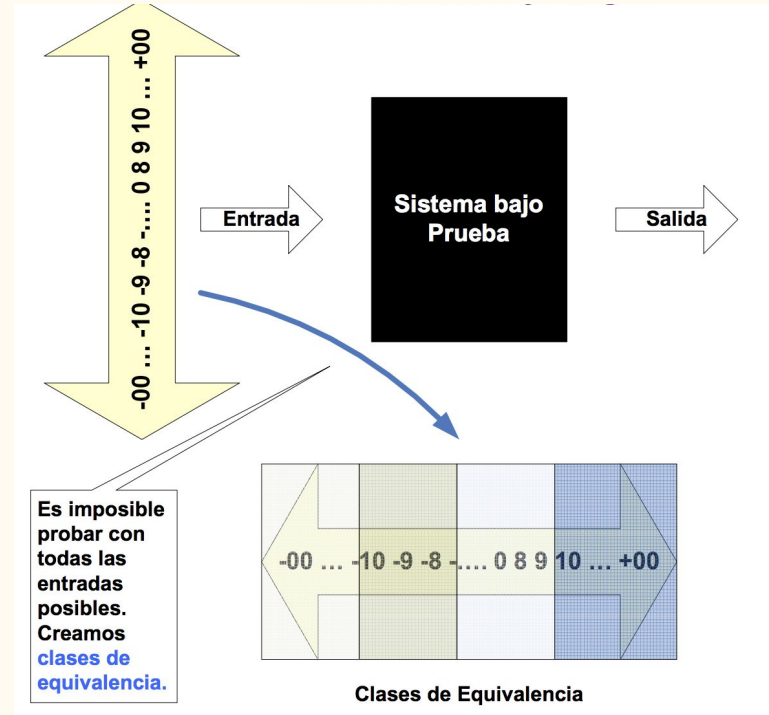




# Caja negra

El Método de la Caja Negra se centra en los requisitos fundamentales del software y permite obtener entradas que prueben todos los requisitos funcionales del programa. Con este tipo de pruebas se intenta encontrar:

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en accesos a las Bases de Datos externas.
- Errores de rendimiento.
- Errores de inicialización y terminación.



# Tipos de pruebas de caja negra

- Métodos gráficos de prueba
- Partición Equivalente
- Análisis de valores límite
- Prueba de tabla ortogonal
- Pruebas de interfaces gráficas
- Prueba de arquitectura cliente/servidor
- Pruebas de servidor
- Pruebas de base de datos
- Pruebas de transacción
- Pruebas de comunicación de red
- Pruebas de documentación
- Pruebas de sistemas de tiempo real

Criterios mínimos que deben guiarnos al escoger los datos de prueba de nuestros programas:

**Valores fáciles,** El programa se depurará con datos de fácil comprobabilidad.

**Valores típicos realistas,** Siempre se ensayará un programa con datos seleccionados para que representen cómo se aplicará.

**Valores extremos,** Valores que desafían el control por ejemplo de ciclos y arreglos.

**Valores ilegales,** "Basura entra, basura sale"

# Herramientas para pruebas de software

JUnit 

 Jira Software

mockito 

# ¿Porqué usar herramientas?

El control de la calidad de software lleva consigo aplicativos que permiten realizar pruebas manuales y autónomas, permitiendo así la verificación desde el punto de vista estático y de caja blanca. Podemos encontrar herramientas escritas en software libre, código abierto o software privativo. Estas herramientas podrán ser utilizadas para diferentes tipos de pruebas como:

- Herramientas de gestión de pruebas
- Herramientas para pruebas funcionales
- Herramientas para pruebas de carga y rendimiento



# Ejemplos

## 1. Herramientas de gestión de pruebas

- [Bugzilla](#) Testopia
- FitNesse
- qaManager
- qaBook
- RTH.<sup>3</sup>
- Salome-tmf
- Squash TM
- Test Environment Toolkit
- TestLink
- Testitool
- XQual Studio
- Radi-testdir
- Data Generator

## 1. Herramientas para pruebas funcionales

- [Selenium](#)
- Soapui
- Watir
- WatIN (Pruebas de aplicaciones web en .Net)
- Capedit
- Canoo WebTest
- Solex
- Imprimatur
- SAMIE
- ITP
- WET
- WebInject

## 1. Herramientas para pruebas de carga y rendimiento

- [JMeter](#)
- Gatling
- FunkLoad
- FWPTT load testing
- loadUI

# JUnit5

JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.

- Pruebas de caja negra.
- Pruebas de regresión.

El propio framework incluye formas de ver los resultados (runners) que pueden ser en modo texto, gráfico (AWT o Swing) o como tarea en Ant.

# Selenium

Selenium es un conjunto de utilidades que facilita la labor de obtener juegos de pruebas para aplicaciones web. Para ello nos permite grabar, editar y depurar casos de prueba, que podrán ser ejecutados de forma automática e iterativa posteriormente.

- Facilidad de registro y ejecución de los test.
- Referencia a objetos DOM en base al ID, nombre o a través de XPath.
- Las acciones pueden ser ejecutadas paso a paso.
- Herramientas de depuración y puntos de ruptura (breakpoints).

# JMeter

JMeter es una herramienta de carga para llevar a cabo simulaciones sobre cualquier recurso de Software.

Inicialmente diseñada para pruebas de estrés en aplicaciones web, hoy en día, su arquitectura ha evolucionado no sólo para llevar a cabo pruebas en componentes habilitados en Internet (HTTP), sino además en Bases de Datos , programas en Perl , requisiciones FTP y prácticamente cualquier otro medio.

# Bugzilla

Bugzilla, es un sistema "de seguimiento de defectos" o "de seguimiento de fallos del sistema". Estos sistemas de seguimiento de defectos permiten a los desarrolladores realizar un seguimiento de los fallos más significativos de su producto de manera eficaz.

Testopia está diseñado para ser una herramienta genérica para el seguimiento de casos de prueba, permitiendo a las organizaciones realizar las pruebas de software e integrar reportes de defectos encontrados, así como el resultado de los de los caso de prueba.

# Automatización



# Vamos a automatizar pruebas

Hoy en día muchas empresas de software, por su negocio quieren ser ágiles.

- Una de las claves para agilizar ese proceso es la optimización y automatización de ciertos procesos y pruebas.
- No es para todos, tiene un coste considerable.
- No suplanta al testing manual.
- La automatización no garantiza la calidad.



# Definición

*“La automatización de pruebas es la práctica que permite controlar la ejecución de un producto software de manera automática, comparando los resultados obtenidos con los resultados esperados”. [3]*

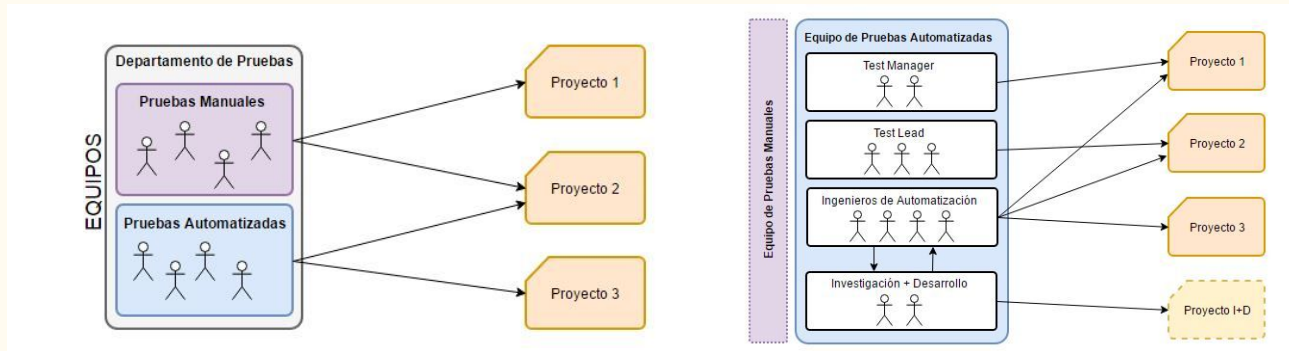
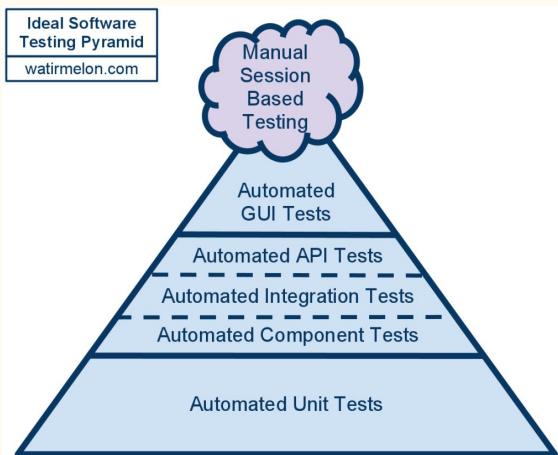


Fig. A. Departamento de Pruebas con pruebas automatizadas [9]

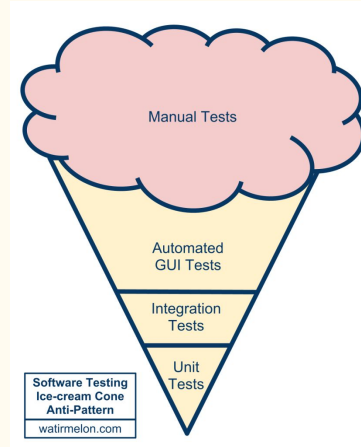
# Pirámide de Cohn



Establece que hay varios niveles de pruebas, y señala el grado en el que deberíamos automatizarlas. [4]

- ❑ Muchos tests unitarios automáticos.
- ❑ Bastantes tests a nivel de API.
- ❑ Menos tests de interfaz gráfica automatizados.
- ❑ Un nivel estable de pruebas automáticas.

# No al Patrón “Cono de Helado”



En la automatización de pruebas hay que tener cuidado, ya que en ciertas ocasiones se tiende a perder el foco y a invertir en automatización en el nivel y grado no adecuado. [5]

# Ejemplos de Aplicaciones



# Sumarizando

Una buena estrategia de automatización de pruebas conlleva más cosas que solo automatizar las pruebas en sí: crear un buen framework de automatización, parametrizar los tests, las ejecuciones para los distintos entornos, lanzar los test con distintos datos de prueba y gestionar dichos datos, sistemas de logs, buenos reportes con información que sirvan para obtener conclusiones, montar una buena infraestructura contra la que lanzar esos tests, paralelizarlos etc.



# Ejercicio: Diseñar pruebas unitarias

En los grupos del proyecto seleccionar dos clases (o componentes unitarios) del proyecto y para cada una:

- Escribir pruebas que validen los atributos (¿Cuándo el valor de un atributo es válido y cuando no?).
- Escribir pares de entradas y salidas esperadas para cada uno de los métodos de acuerdo a las estrategias de selección de pruebas unitarias.

Hacerlo para al menos tres atributos y tres métodos no triviales de cada clase.



# Ejercicio: Preguntas

En los grupos del proyecto discuta lo siguiente:

- Plantee la posibilidad de usar una prueba de versión, en que escenario lo utilizarías?, argumente.
- Elabore una prueba basada en requerimientos a un requisito de tu sistema, considere todos los aspectos y características posibles para el mismo.





# Referencias

- [1] -- Guru99 What is software testing? Introduction, definition & types. Disponible en: <https://bit.ly/2ZJi6dw>
- [2] -- I. Sommerville. Ingeniería de Software, 9th ed. México: PEARSON EDUCACIÓN, 2011.
- [3] -- A. Crespo. (2018, Jan 11). Automatización de pruebas [Online]. Available: <https://bit.ly/2UJW5aT>
- [4] -- E. S. Alberola (2017, Feb 22). ¿Cómo automatizar pruebas y no morir en el intento? [Online]. Available: <https://bit.ly/2V2A7F6>
- [5] -- A. M. del C. García. (2015, Jan 23). “Vamos a automatizar pruebas”. ¿Qué significa esto? ¿Realmente por dónde deberíamos empezar a automatizar? [Online]. Available: <https://bit.ly/2vrBYnE>