# Importancia de las pruebas de software

Dr. Eduardo A. Rodríguez Tello

Cinvestav-Tamaulipas

30 de marzo del 2011

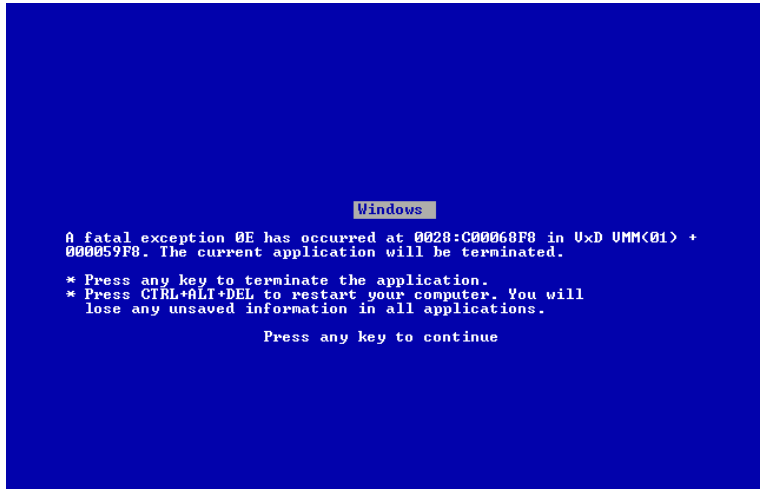
1. Importancia de las pruebas de software
   - Introducción
   - Conceptos básicos
   - Niveles de pruebas
   - Métodos de prueba

# Introducción

- En un proyecto de desarrollo de software los errores (*bugs* en inglés) puede presentarse en cualquiera de las etapas del ciclo de vida del software
- Aún cuando se intente detectarlos despúes de cada fase utilizando técnicas como la inspección, algunos errores permanecen sin ser descubiertos

# Introducción

- Por lo tanto es muy probable que el código final contenga errores de requerimientos y diseño, adicionales a los introducidos en la codificación


# Introducción

- Las *pruebas de software* son una parte importante pero muy *costosa* del proceso de desarrollo de software
- Pueden llegar a representar entre el 30 y 50 % del costo total del desarrollo del software [Myers, 2004]
- Sin embargo, los costos de las fallas en un software en operación pueden llegar a ser mucho mayores (catastróficos)

# Introducción

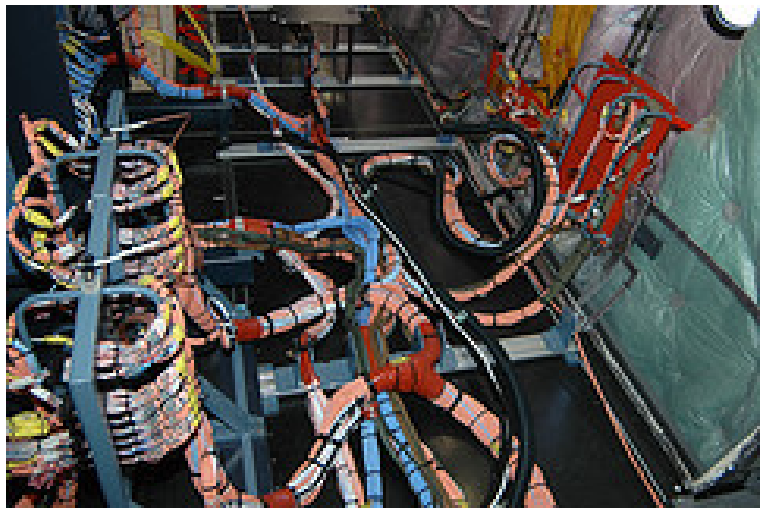
## Algunos de los peores errores de la historia

- Se colapsa el aeropuerto de Los Angeles (2007)
- Más de 17 mil personas se quedaron en tierra por un problema de software que provocó conflictos en una tarjeta de red que bloqueó toda la red informática

# Introducción

## Algunos de los peores errores de la historia

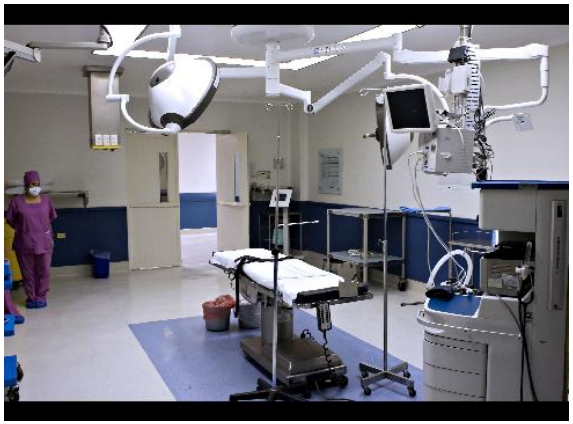
- El lanzamiento comercial y la producción del *Airbus A380* se retrasa más un año (2006)
- Diferencias entre versiones de las herramientas CAD (Computer Aided Design) usadas en las fábricas de Hamburgo y Toulouse provocaron un problema en el cableado (530km de cables)

# Introducción

## Algunos de los peores errores de la historia

- Sobredosis radiológica en el Instituto Nacional del Cáncer de Panama (2000)
- Errores en los procedimientos y un fallo de software causan que se apliquen dosis erróneas de radiación
- 8 personas murieron y 20 tuvieron problemas de salud graves. Los médicos responsables del hecho fueron acusados de asesinato



# Introducción

- Como pueden observar las *pruebas de software* tienen un rol muy importante en el aseguramiento de la calidad ya que permiten detectar los errores introducidos en las fases previas del proyecto
- Durante esta platica analizaremos algunas de las estrategias y técnicas más importantes para efectuar las pruebas de software

# Conceptos básicos

## Proceso de pruebas

- La forma más común de organizar las actividades relacionadas al proceso de pruebas de software [Burnstein, 2003] son:
  - *Planeación*, fija las metas y una estrategia general de pruebas
  - *Preparación*, se describe el procedimiento general de pruebas y se generan los casos de prueba específicos
  - *Ejecución*, incluye la observación y medición del comportamiento del producto
  - *Análisis*, incluye verificación y análisis de resultados para determinar si se observaron fallas
  - *Seguimiento*, si se detectaron fallas, se inicia un monitoreo para asegurar que se remueva el origen de éstas

# Conceptos básicos

## Casos de prueba y criterios de prueba

- Generar casos de prueba *efectivos* que revelen la presencia de fallas es fundamental para el éxito del proceso de pruebas (etapa de preparación)
- Idealmente, se debería determinar un conjunto de casos de prueba tales que su ejecución exitosa implique que no hay errores en el software desarrollado
- Comúnmente este objetivo ideal no se puede lograr debido a las limitaciones prácticas y teóricas

# Conceptos básicos

## Casos de prueba y criterios de prueba...

- Cada caso de prueba cuesta dinero: esfuerzo para generarlo, tiempo de cómputo para ejecutarlo, esfuerzo para evaluar los resultados
- Por lo tanto, el número de casos de prueba necesarios para detectar los errores debe ser minimizado para reducir costos

Cinvestav

# Conceptos básicos

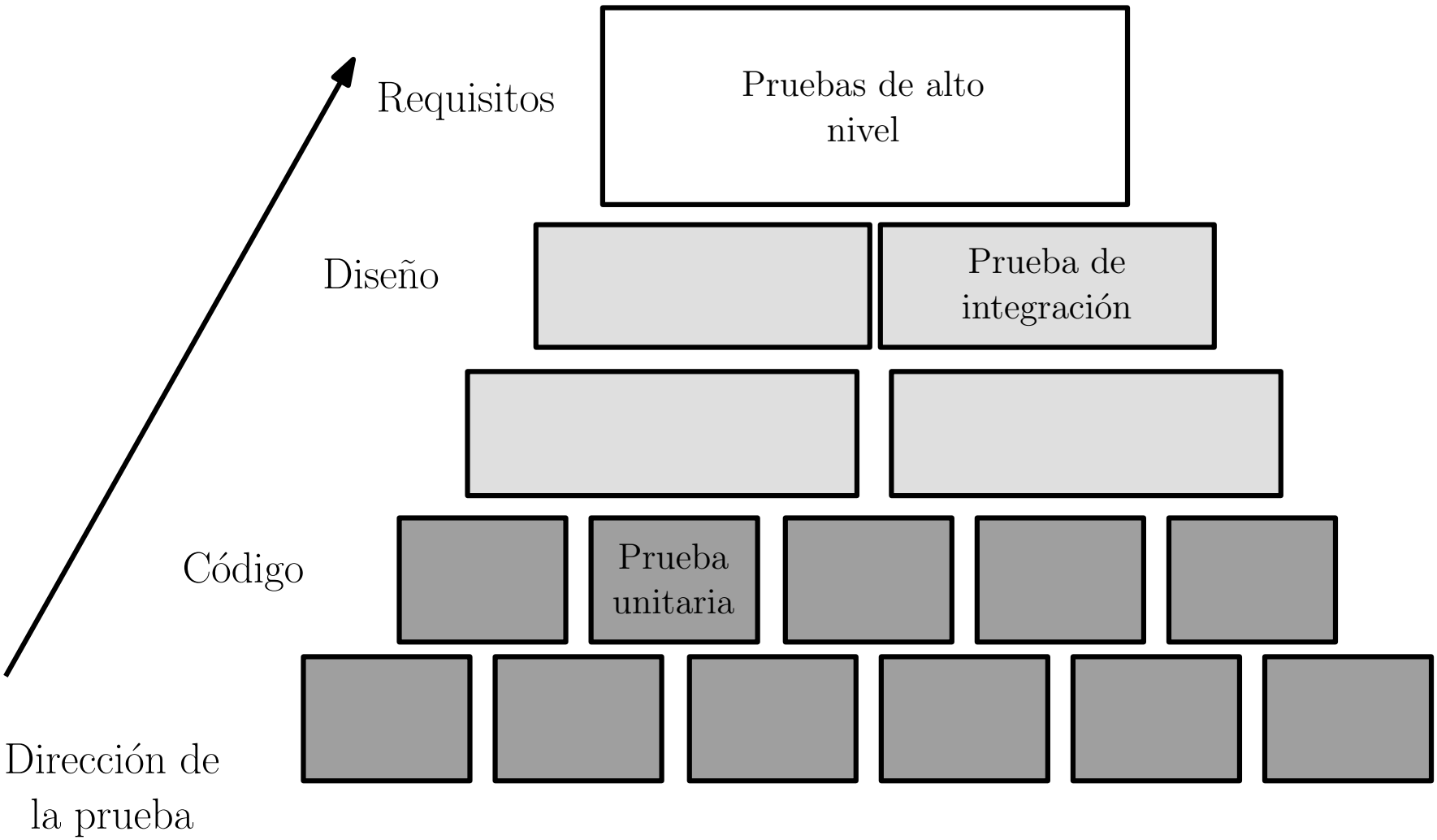
## Objetivos del proceso de pruebas

- Los dos objetivos principales del proceso de pruebas:
  - Maximizar el número de errores detectados (cobertura)
  - Reducir al mínimo el número de casos de prueba (costo)
- Como con frecuencia son contradictorios, el problema de seleccionar el conjunto de casos de prueba con el que un programa debe ser probado se vuelve una tarea muy compleja

# Niveles de pruebas

- Generalmente se comienza probando las partes más pequeñas y se continua con las más grandes
- Para el software convencional
  - El módulo (componente) se prueba primero
  - Se continua con la integración de módulos
- Para el software orientado a objetos
  - Se prueba primero una clase (atributos, métodos, colaboración)

# Niveles de pruebas



# Pruebas unitarias

- Se concentran en probar cada componente individualmente para asegurar que funcione de manera apropiada como unidad
- Emplean técnicas de prueba que recorren caminos específicos en la estructura de control de los componentes (pruebas estructurales)

# Pruebas unitarias

- Herramientas
  - JUnit
  - TestNG (versión mejorada de JUnit)
  - PHPUnit
  - CPPUnit
  - NUnit (.Net)
  - MOQ (creación dinámica de objetos simuladores, *mocks*)

# Pruebas de integración

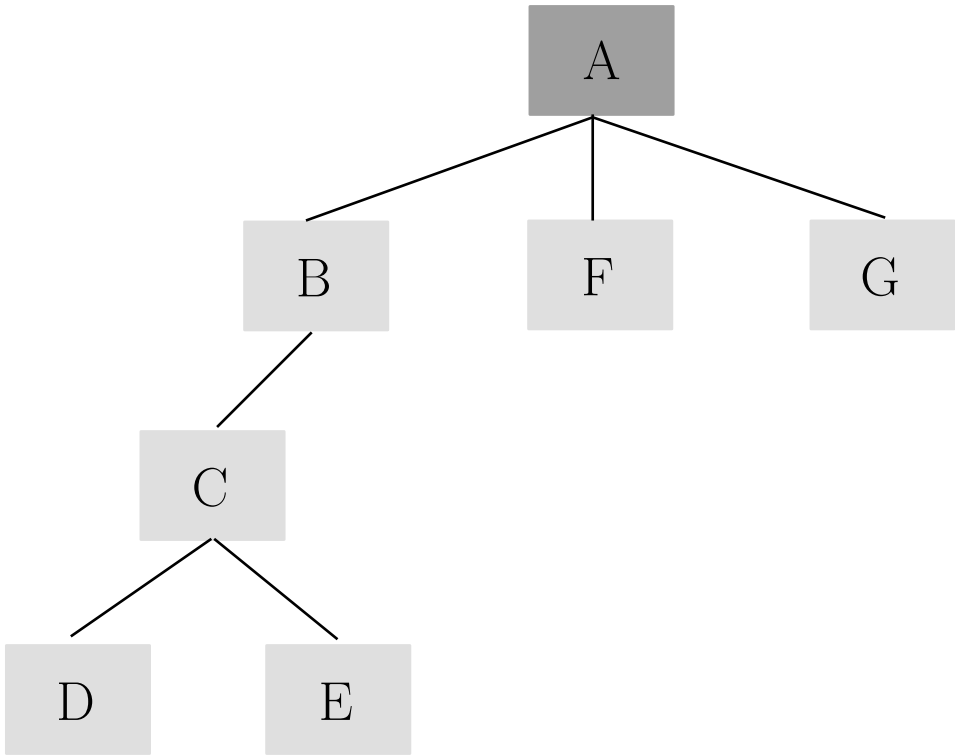
- Las pruebas de integración tienen dos objetivos principales:
  - Descubrir errores asociados con las interfaces de los módulos
  - Ensamblar sistemáticamente los módulos individuales para formar subsistemas y al final un sistema completo
- Principalmente se utilizan técnicas que verifican el correcto manejo de las entradas y salidas del software (pruebas funcionales)
- También pueden emplearse técnicas que recorren rutas específicas en la estructura de control del software (pruebas estructurales)

# Pruebas de integración

- Existen dos enfoques principales para las pruebas de integración (incremental):
  - Integración descendente (componentes de funcionales)
  - Integración ascendente (componentes de infraestructura, e.g. acceso a BD)

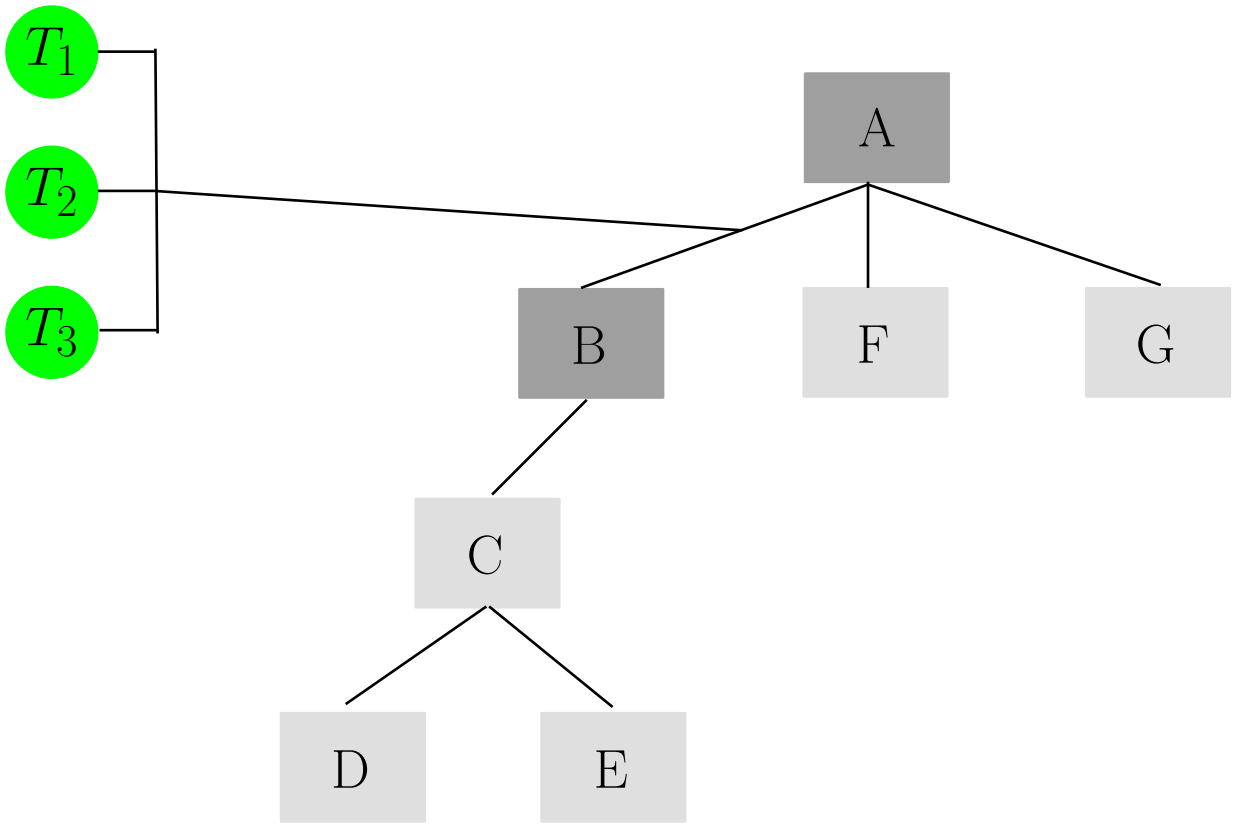
# Pruebas de integración, enfoque descendente

- El módulo principal es usado como controlador y todos sus módulos subordinados son remplazados por módulos simulados (*stubs*)



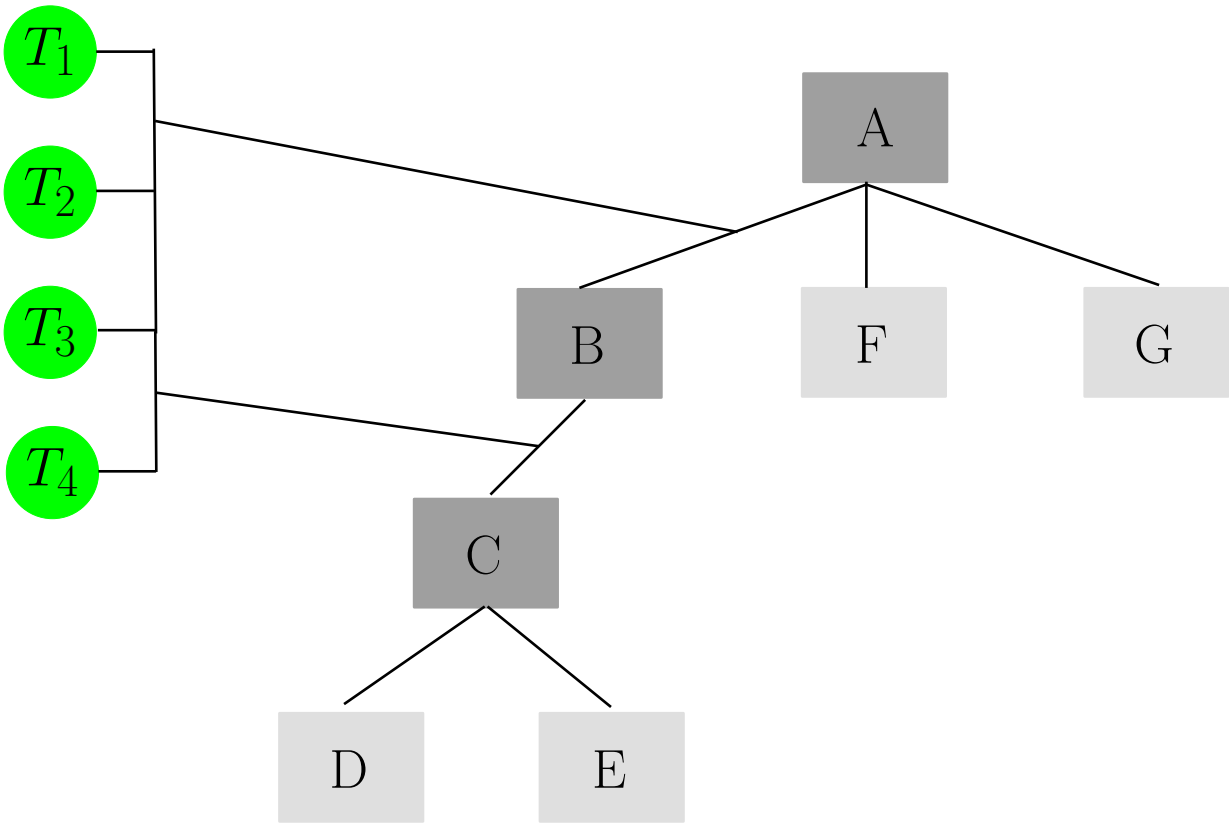
# Pruebas de integración, enfoque descendente

- Los módulos simulados se remplazan uno a la vez con los componentes reales (en profundidad) y se van probando



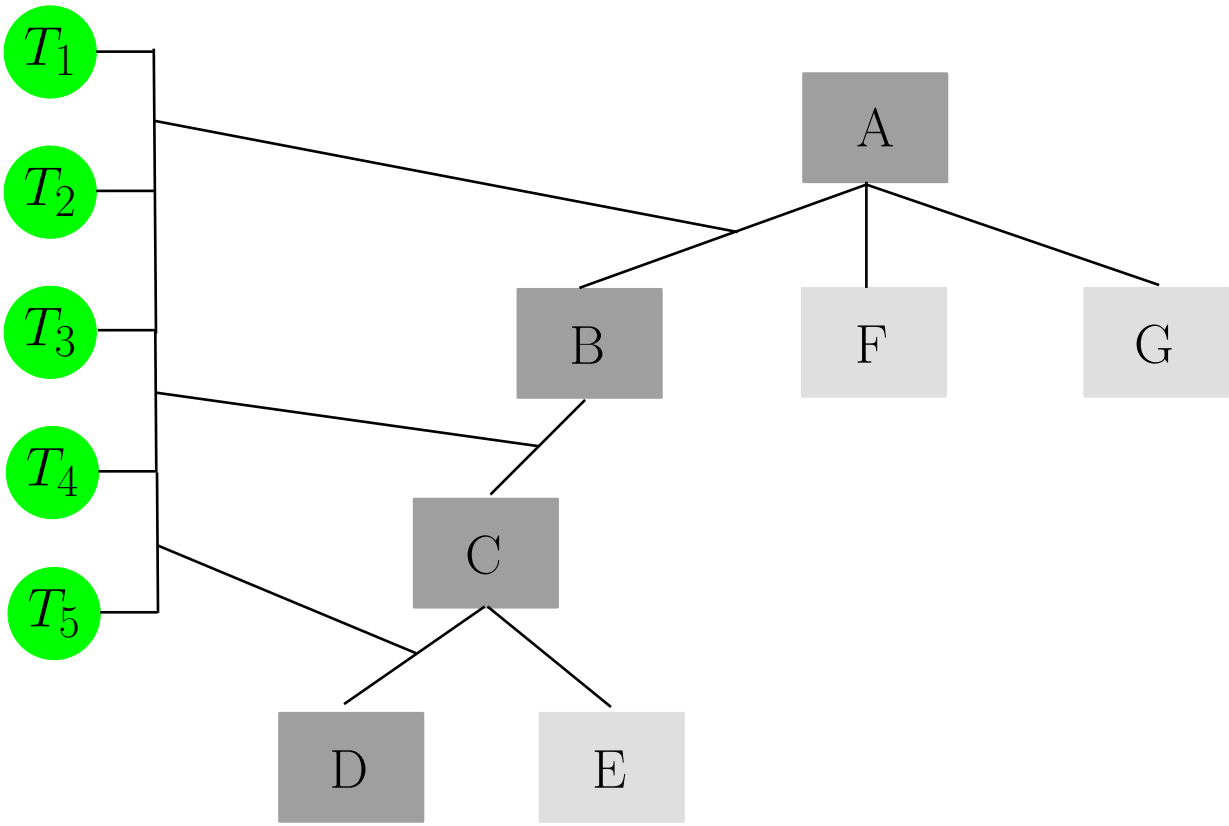
# Pruebas de integración, enfoque descendente

- Los módulos simulados se remplazan uno a la vez con los componentes reales (en profundidad) y se van probando



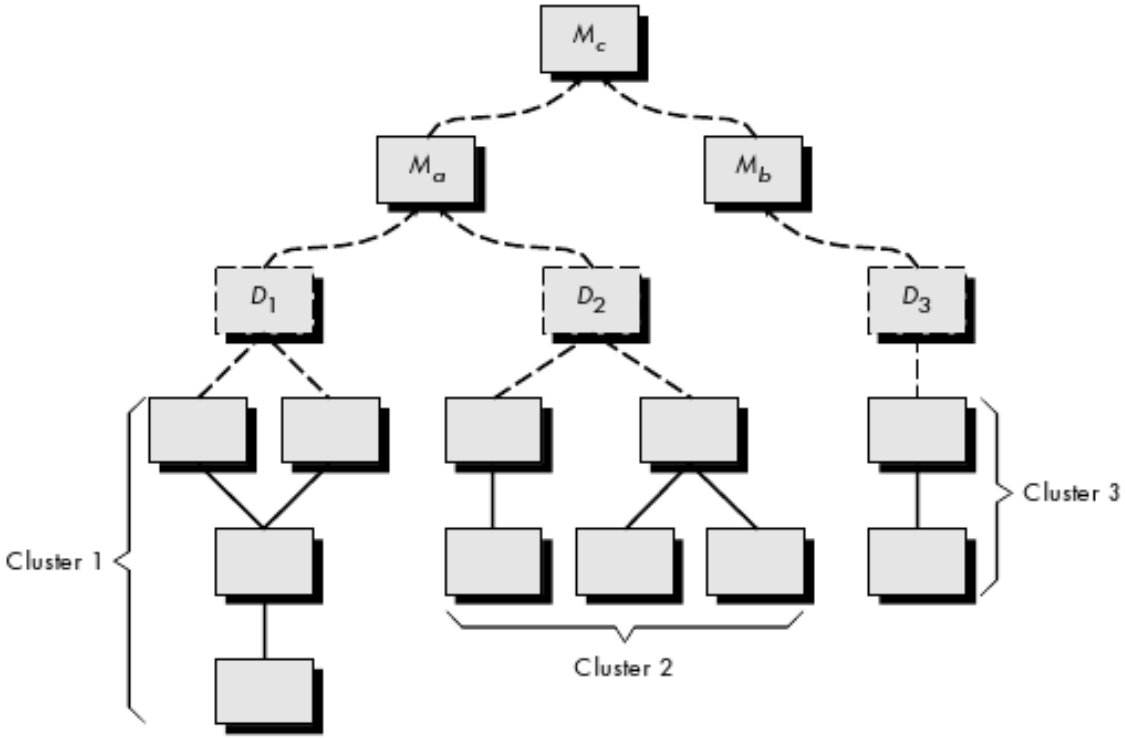
# Pruebas de integración, enfoque descendente

- Los módulos simulados se remplazan uno a la vez con los componentes reales (en profundidad) y se van probando



# Pruebas de integración, enfoque ascendente

- A diferencia del enfoque descendente, éste inicia la construcción y prueba de los módulos en los niveles más bajos de la estructura del programa
- No se requiere el uso de módulos simulados (*stubs*)



# Pruebas de alto nivel

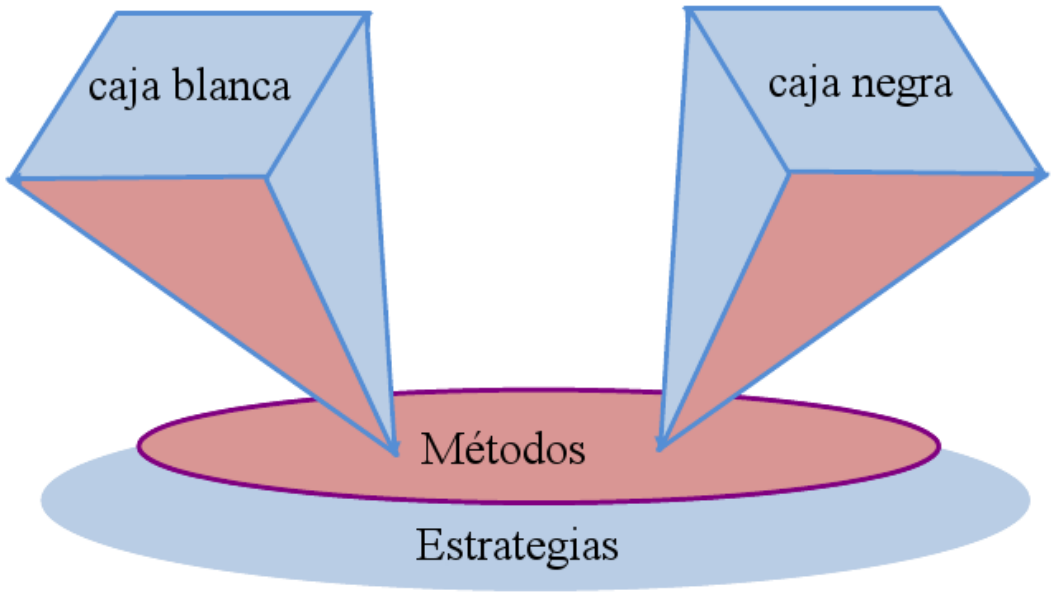
- *Pruebas de validación*, se enfocan en los requerimientos
  - Pruebas de aceptación: desarrolladas por el cliente
  - Pruebas alfa: realizadas por el usuario con el desarrollador como observador
  - Pruebas beta: realizadas por el usuario en su entorno de trabajo (sin observadores)

# Pruebas de alto nivel

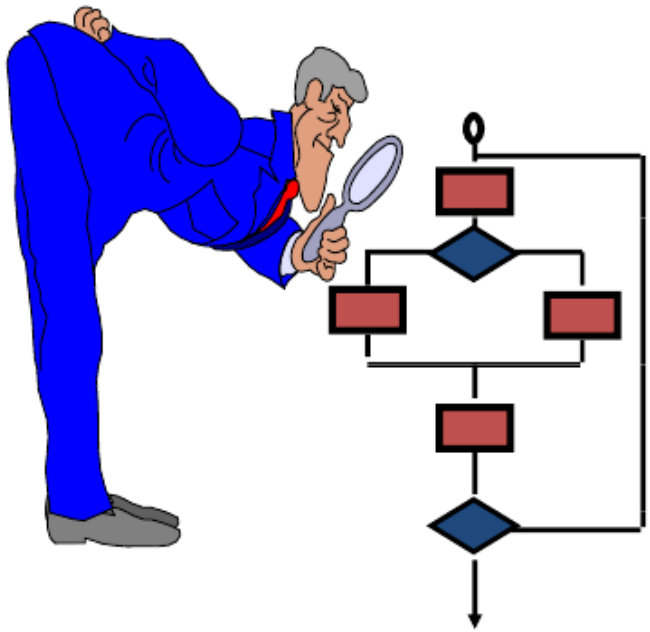
- *Pruebas del sistema*, se enfocan en la integración del sistema (Hw, información, personas)
  - *Prueba de recuperación*, forza el software a fallar en diferentes formas y verifica que éste se recupere adecuadamente
  - *Prueba de seguridad*, verifica que los mecanismos de protección integrados en el sistema realmente impidan irrupciones inapropiadas
  - *Prueba de resistencia*, ejecutan un sistema de manera que se demanden recursos en cantidades, frecuencias o volúmenes anormales
  - *Prueba de desempeño*, prueba el desempeño del software en tiempo de ejecución dentro del contexto de un sistema integrado

# Métodos de prueba

- Existen dos métodos básicos para diseñar casos de prueba
  - De *caja blanca* (o estructurales)
  - De *caja negra* (o funcionales)



# Métodos de prueba, de caja blanca



- Verifican la correcta implementación de las unidades internas, las estructuras y sus relaciones
- Hacen énfasis en la reducción de errores internos

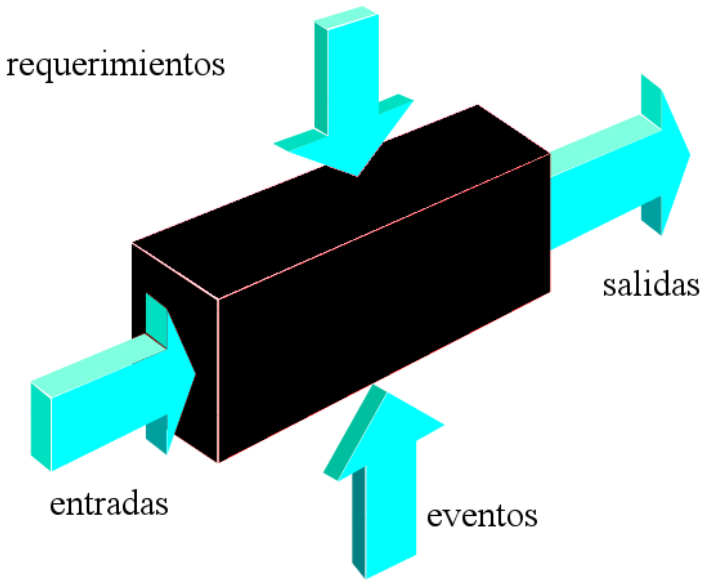
# Métodos de prueba, de caja blanca

- Los métodos de caja blanca o estructurales permiten derivar casos de prueba que
  - Garanticen que todas las rutas independientes dentro del módulo se ejecuten al menos una vez
  - Ejecuten los lados verdadero y falso de todas las decisiones lógicas
  - Ejecuten todos los ciclos dentro y en sus límites operacionales
  - Ejerciten las estructuras de datos internas para asegurar su validez

# Métodos de prueba, de caja blanca

- Algunos ejemplos de técnicas de caja blanca
  - Prueba de ruta básica
    - Complejidad ciclomática
  - Pruebas de estructura de control
    - Prueba de condición
    - Prueba de flujo de datos
    - Prueba de ciclos simples
    - Prueba de ciclos anidados

# Métodos de prueba, de caja negra




- Verifican el correcto manejo de funciones externas provistas o soportadas por el software
- Verifican que el comportamiento observado se apegue a las especificaciones del producto y a las expectativas del usuario
- Los casos de prueba se construyen a partir de las especificaciones del sistema

Cinvestav

# Métodos de prueba, de caja negra

- Los métodos de caja negra o funcionales permiten derivar casos de prueba que buscan encontrar los siguientes tipos de errores:
  - Funciones incorrectas o faltantes
  - Errores de interfaz
  - Errores en estructuras de datos o en acceso a BD externas
  - Errores de comportamiento o desempeño
  - Errores de inicialización o término

# Métodos de prueba

## Prueba exhaustiva

- El procedimiento de prueba de caja negra más obvio es la *prueba exhaustiva*

| Hardware | Browser | OS | Connection | Memory |
|---|---|---|---|---|
| PC | IE | Windows | LAN | 1GB |
| Laptop | Mozilla | Linux | ISDN | 2GB |

- Para probar exhaustivamente este sistema con 5 componentes (parámetros) cada uno con 2 valores se requieren ejecutar $2^5 = 32$ configuraciones diferentes (casos de prueba)

# Métodos de prueba

## Prueba exhaustiva

- Para un sistema simple como el de nuestro ejemplo es posible ejecutar una prueba exhaustiva
- Sin embargo, este enfoque es impractico y no factible por que el número de casos de prueba crece muy rápidamente
- Por ejemplo para probar exhaustivamente un sistema con 5 parámetros cada uno con 10 valores se requieren ejecutar $10^5 = 1000000$ casos de prueba
- Si se ejecuta y evalua un caso de prueba por minuto tardaríamos 694.44 días
- ¿Existen otras alternativas?

# Métodos de prueba

## Estrategias combinatorias, pruebas de interacción

- Sí, las *pruebas de interacción* (ver [Grindal et al., 2005])
- Este enfoque identifica fallas que surgen de las interacciones de $t$ (o menos) parámetros de entrada
- Para ello crea casos de prueba que incluyen al menos una vez todas las $t$-combinaciones entre estos parámetros y sus valores
- Los Covering arrays (CAs) son objetos combinatorios usados para representar pruebaa de interacción [Hartman, 2005]
- Permiten maximizar el número de errores detectados reduciendo al mínimo el número de casos de prueba [Kuhn et al., 2004]

# Métodos de prueba

## Ejemplo de covering arrays CA(10; 3, 5, 2). Prueba exhaustiva $2^5 = 32$, CAs 10

| Hardware | Browser | OS | Connection | Memory |
|---|---|---|---|---|
| PC | IE | Windows | LAN | 1GB |
| Laptop | Mozilla | Linux | ISDN | 2GB |

| Test | Hardware | Browser | OS | Connection | Memory |
|---|---|---|---|---|---|
| 1 | Laptop | Mozilla | Windows | ISDN | 2GB |
| 2 | Laptop | IE | Windows | ISDN | 1GB |
| 3 | PC | IE | Windows | ISDN | 2GB |
| 4 | Laptop | IE | Linux | ISDN | 2GB |
| 5 | Laptop | IE | Windows | LAN | 2GB |
| 6 | PC | IE | Linux | LAN | 1GB |
| 7 | Laptop | Mozilla | Linux | LAN | 1GB |
| 8 | PC | Mozilla | Windows | LAN | 1GB |
| 9 | PC | Mozilla | Linux | LAN | 2GB |
| 10 | PC | Mozilla | Linux | ISDN | 1GB |

# Importancia de las pruebas de software

Dr. Eduardo A. Rodríguez Tello

http://www.tamps.cinvestav.mx/~ertello
ertello@tamps.cinvestav.mx

30 de marzo del 2011

# Referencias I


Burnstein, I. (2003).

*Practical Software Testing*.
Springer, New York, USA.

Grindal, M., Offutt, J., and Andler, S. (2005).

Combination testing strategies: a survey.
*Software Testing, Verification, and Reliability*, 15(3):167–199.

Hartman, A. (2005).

Software and hardware testing using combinatorial covering suites.
In *Graph Theory, Combinatorics and Algorithms*, chapter 10, pages 237–266. Springer-Verlag.

Kuhn, D. R., Wallace, D. R., and Gallo, A. M. (2004).

Software fault interactions and implications for software testing.
*IEEE Transactions on Software Engineering*, 30(6):418–421.

Myers, G. J. (2004).

*The art of software testing*.
Wiley, second edition.