Estrategias y técnicas de prueba del software

Dr. Eduardo A. Rodríguez Tello

Cinvestav-Tamaulipas

24 de octubre del 2012

1. Estrategias y técnicas de prueba del software
   - Introducción
   - Conceptos básicos
   - Niveles de pruebas
   - Depuración
   - Métodos de prueba
   - Pruebas de caja blanca
   - Pruebas de caja negra

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

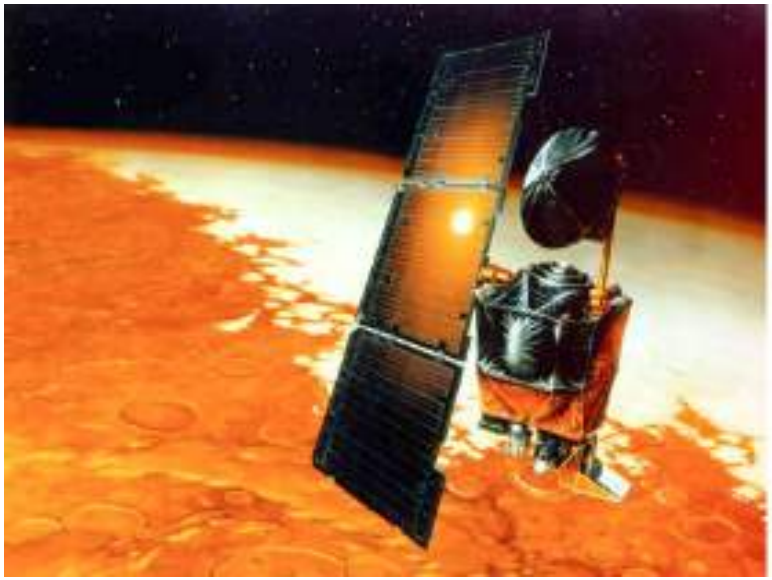
Algunos de los peores errores de la historia

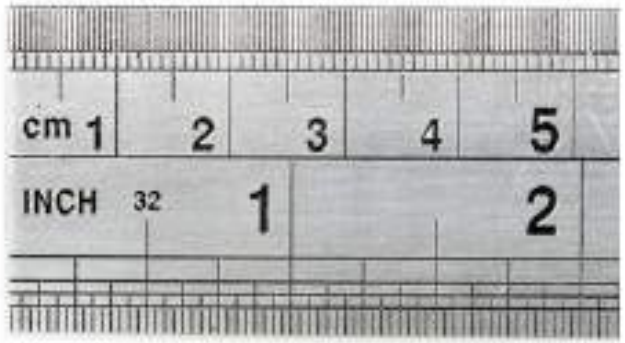
- Falla de un misil antibalístico *Patriot* durante la primera guerra del Golfo Pérsico (1991)
- Errores en el software de cálculo de trayectorias y activación del misil causaban fallas en algunas ocasiones
- 28 soldados muertos en una base Arabia Saudita (Dhahran) a causa de un ataque con misiles *Scud*

Introducción

Algunos de los peores errores de la historia

- Desintegración de una sonda climática enviada a Marte por la NASSA (1999)
- Errores en el software de navegación causan que se estrelle en la atmósfera marciana (incorrecto uso de unidades métricas)
- Después de 9 meses de viaje y un gasto de 125 millones de dolares el proyecto fracasó





Introducción

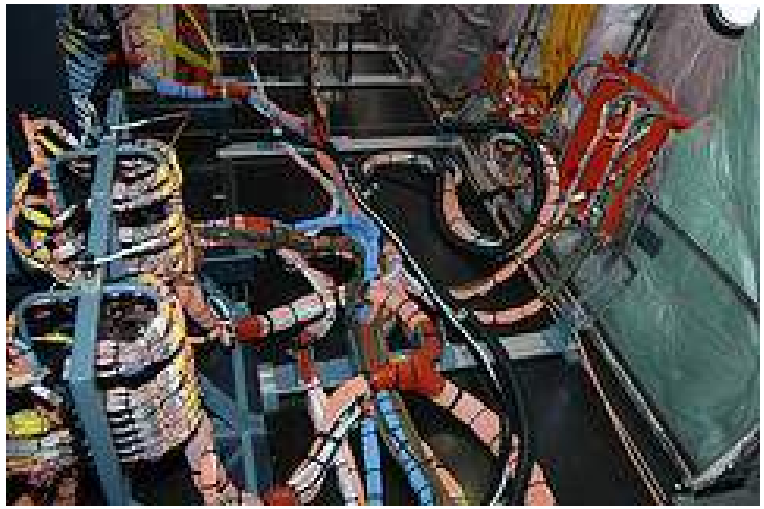
Algunos de los peores errores de la historia

- 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

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

- 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

- 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 este seminario analizaremos a detalles algunas de las estrategias y técnicas más importantes para efectuar las pruebas de software

Conceptos básicos
Definiciones

- **Error**, se refiere a la diferencia entre la salida actual de un software y la correcta. También puede ser una acción que introduce en el software un defecto o falla

$$(1 + \text{sen}(90))^2 =$$

Conceptos básicos
Definiciones

- **Error**, se refiere a la diferencia entre la salida actual de un software y la correcta. También puede ser una acción que introduce en el software un defecto o falla

$$(1 + \text{sen}(90))^2 = \mathbf{5}$$

Conceptos básicos
Definiciones

- **Defecto**, es una condición que causa que un software falle al realizar una función requerida (sinónimo de *bug*)

Función Promedio

Recibe como entrada un vector A de tamaño n
$suma \leftarrow 0$
for $(i \leftarrow 0; i < n; i{+}{+})$ **do**
  $suma \leftarrow suma + A[i]$
end for
return (suma / 10)

Conceptos básicos
Definiciones

- **Falla**, es la incapacidad de un software para realizar una función requerida de acuerdo con sus especificaciones (son producidas por defectos)

Función Promedio

Entrada vector $A = \{10, 8, 9, 10, 9\}$ de tamaño $n = 5$
$suma \leftarrow 0$
for $(i \leftarrow 0; i < n; i{+}{+})$ **do**
  $suma \leftarrow suma + A[i]$
end for
return (suma / 10)

- Regresa como resultado 4.6 en vez de 9.2

Conceptos básicos
Definiciones

- **Caso de prueba**, está dado por un conjunto de entradas, condiciones de ejecución y las salidas esperadas, permite revelar fallas

Función Promedio

Recibe como entrada un vector A de tamaño n
$suma \leftarrow 0$
for $(i \leftarrow 0; i < n; i{+}{+})$ **do**
  $suma \leftarrow suma + A[i]$
end for
return $(suma/n)$

- Caso de prueba 1:
  - Entrada: $A = \{10, 10, 10, 10, 10, 10\}$, $n = 6$
  - Salida: 10.0

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

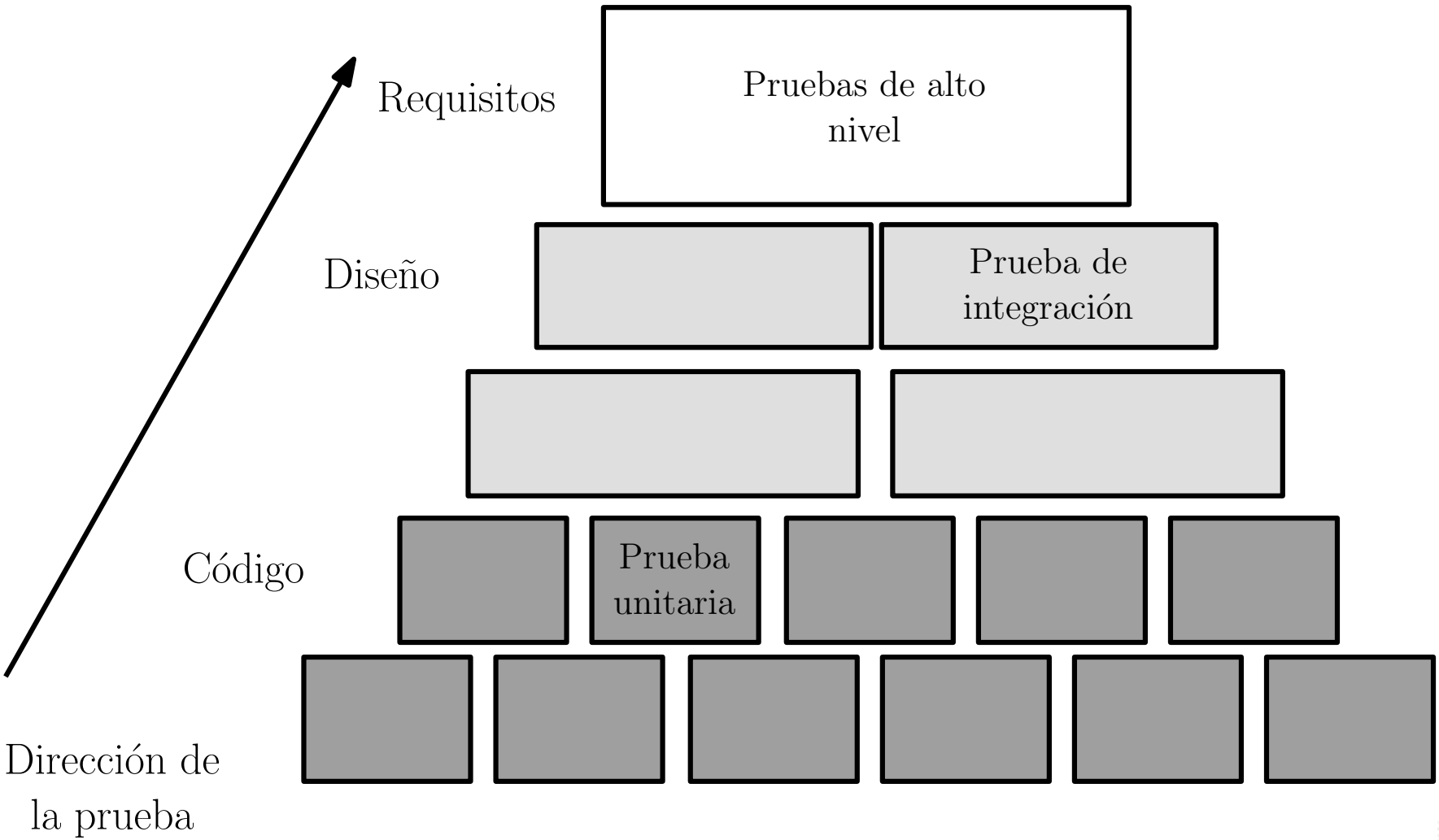
Conceptos básicos
Objetivos del proceso de pruebas

- 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)

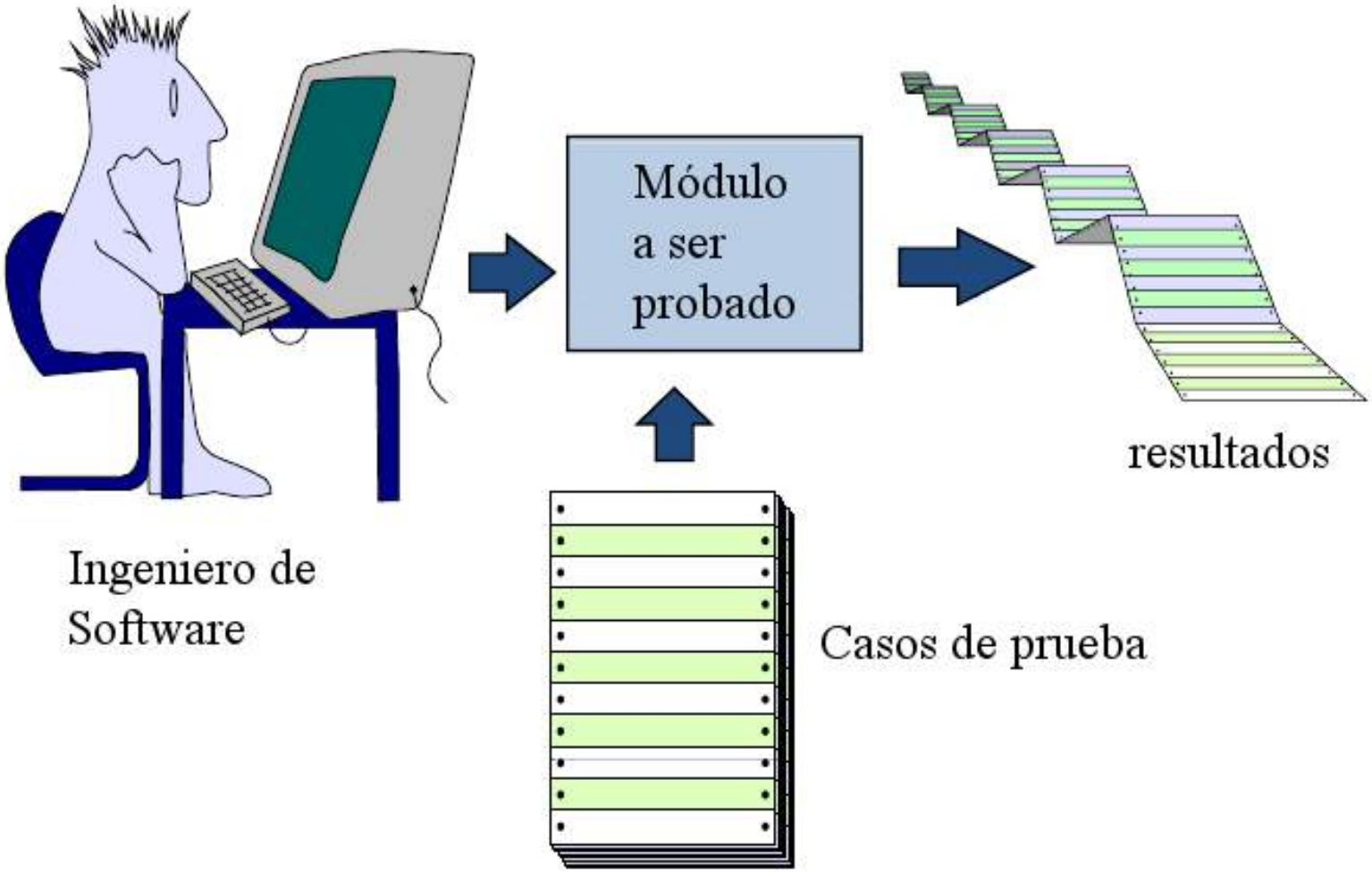
Niveles de pruebas



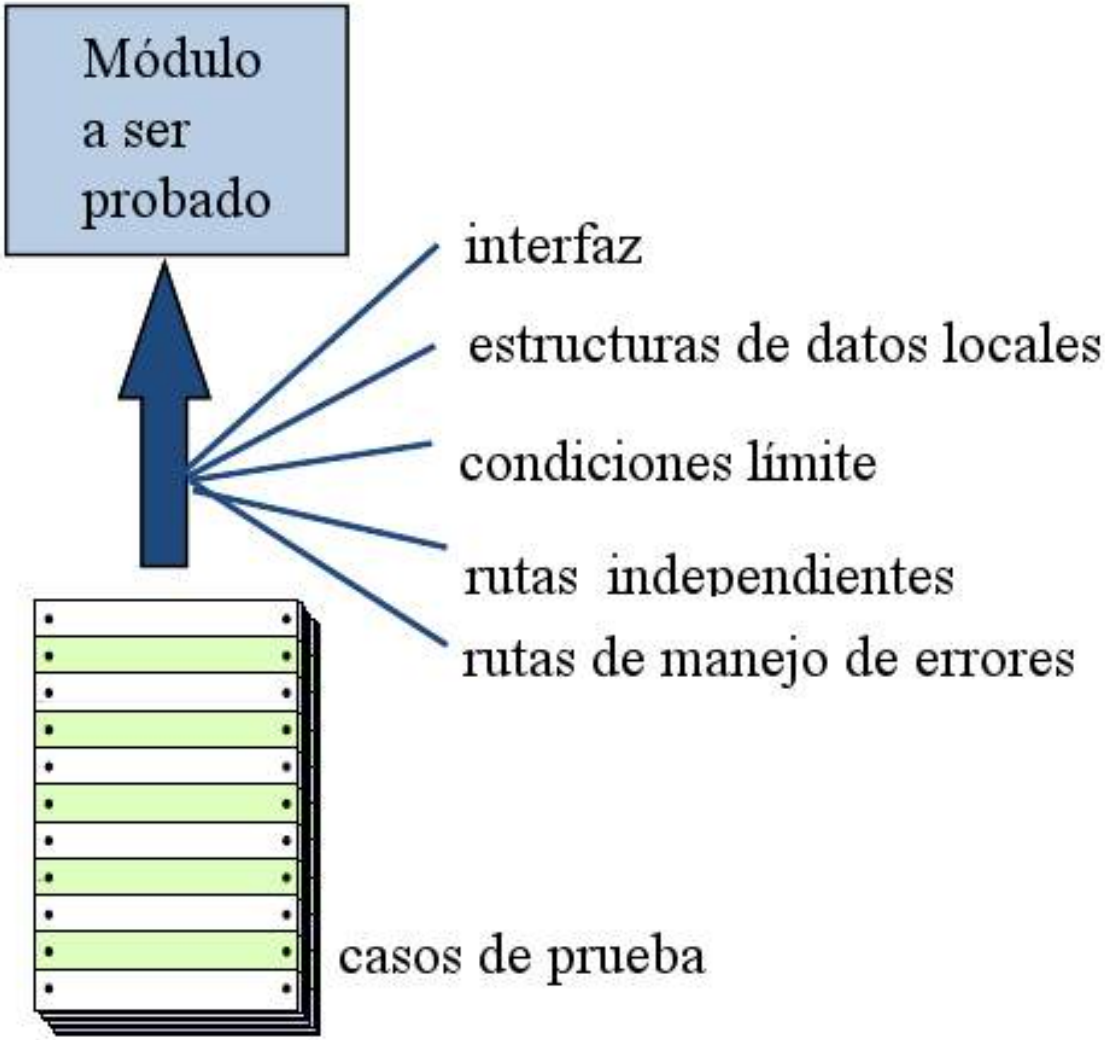
Pruebas unitarias

- 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

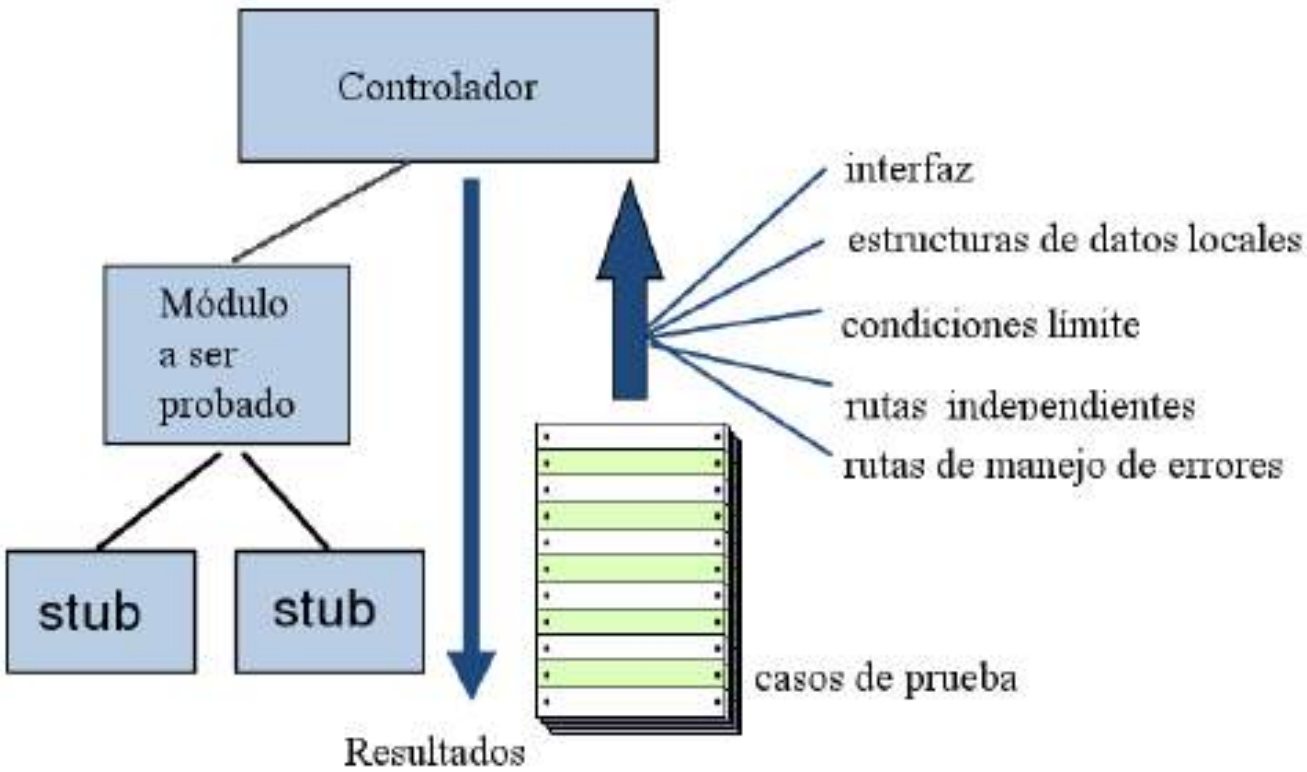


Pruebas unitarias



Pruebas unitarias
Ambiente

- Para probar cada módulo se requiere de un *módulo controlador* y de uno o más *módulos simulados* (*stubs*)



Pruebas unitarias

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

Breve introducción a JUnit 4

- Una herramienta libre para escribir pruebas unitarias en Java
- `www.junit.org`
- Soporta la ejecución de pruebas en modo *batch* o mediante *GUI*
- Los casos de prueba pueden agruparse (*test suites*)
- Todos las pruebas son del tipo *pasa/falla*
- Los mensajes de error y la pila de ejecución indican donde pudo originarse el error

Breve introducción a JUnit 4

- Para probar una clase se escribe una clase de prueba
- Cada prueba unitaria está compuesta de un método de prueba
- Los casos de prueba se identifican con la anotación **@Test**
- Los métodos de inicialización y finalización se identifican con la anotación **@Before** y **@After** (son llamados antes y después de cada método de prueba)

Breve introducción a JUnit 4

- Los métodos de inicialización y finalización que aplican a la clase de prueba entera se identifican con **@BeforeClass** y **@AfterClass** (se ejecutan sólo una vez)
- Es posible verificar excepciones en los métodos de prueba: **@Test(expected=IllegalArgumentException.class)**
- Se puede especificar un tiempo máximo permitido (milisegundos) para ejecutar un método de prueba: **@Test(timeout=1000)**

Breve introducción a JUnit 4

- Tipos de assert:
  - assert(boolean condition)
  - assertTrue(boolean condition)
  - assertFalse(boolean condition)
  - assertEquals(boolean condition)
  - assertNotNull(Object)
  - assertNull(Object)
- fail(String message)

Breve introducción a JUnit 4

- Veamos un ejemplo práctico ...

Pruebas de integración

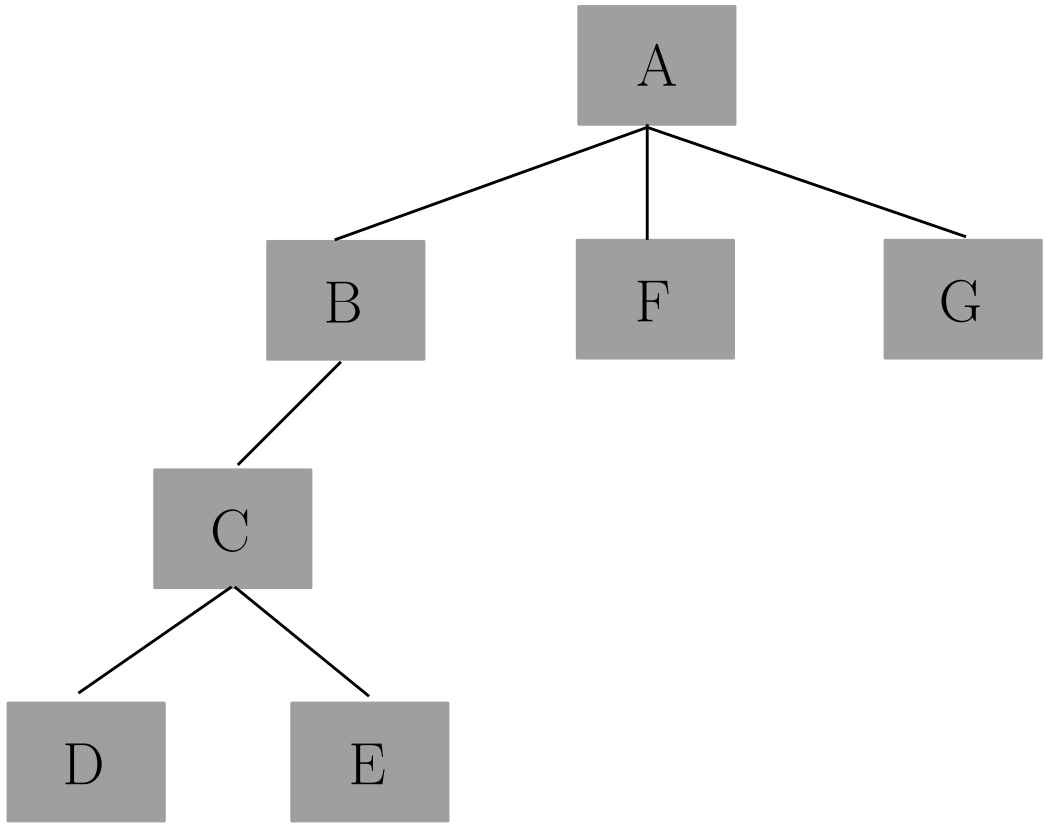
- Las pruebas de integración tienen dos objetivos principales:
  - Descubrir errores asociados con las interfaces de los módulos
  - Ensamblar sistemáticamente los módulos individuales para formar subsistemas y al final un sistema completo
- Principalmente se utilizan técnicas que verifican el correcto 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
  - *Big bang*, combinar todos los componentes y probar el sistema como un todo
  - *Integración incremental*, los componentes se integran y prueban poco a poco
    - Integración descendente (componentes de funcionales)
    - Integración ascendente (componentes de infraestructura, e.g. acceso a BD)

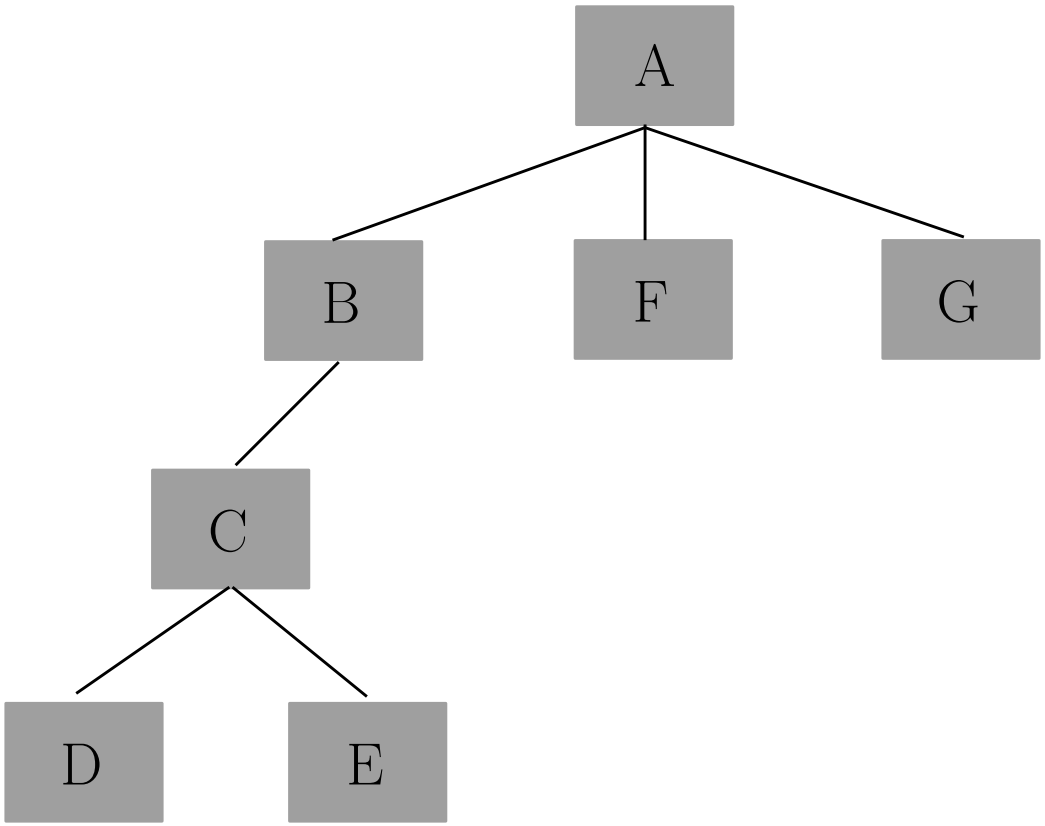
Pruebas de integración, descendente

- Para explicar este enfoque utilizaremos el siguiente grafo de llamadas (diagrama de estructura)



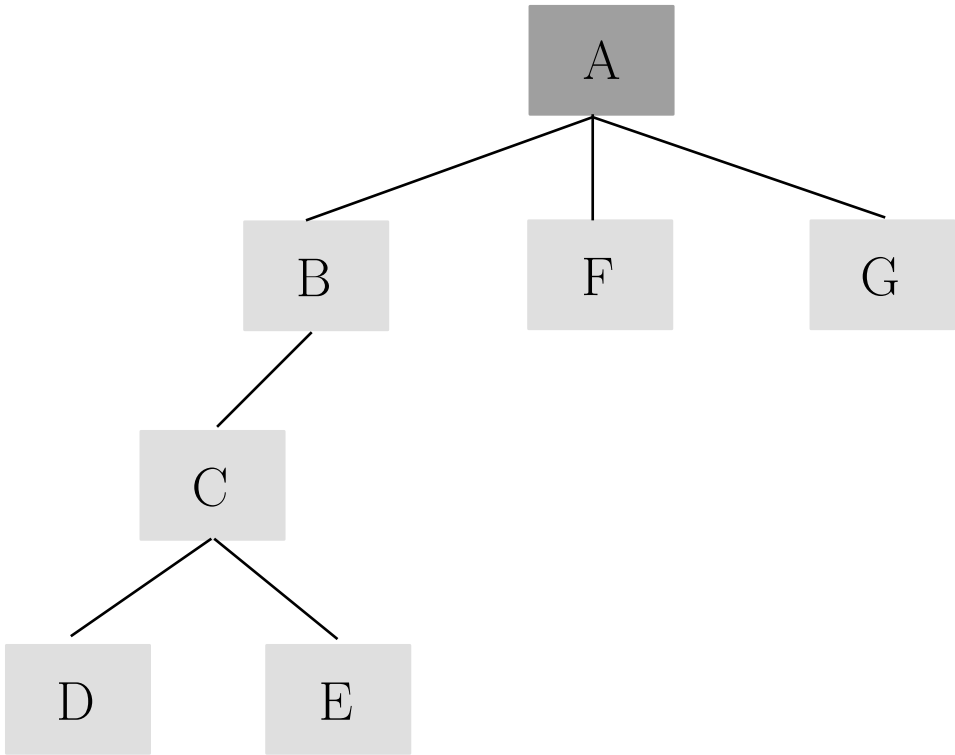
Pruebas de integración, descendente

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



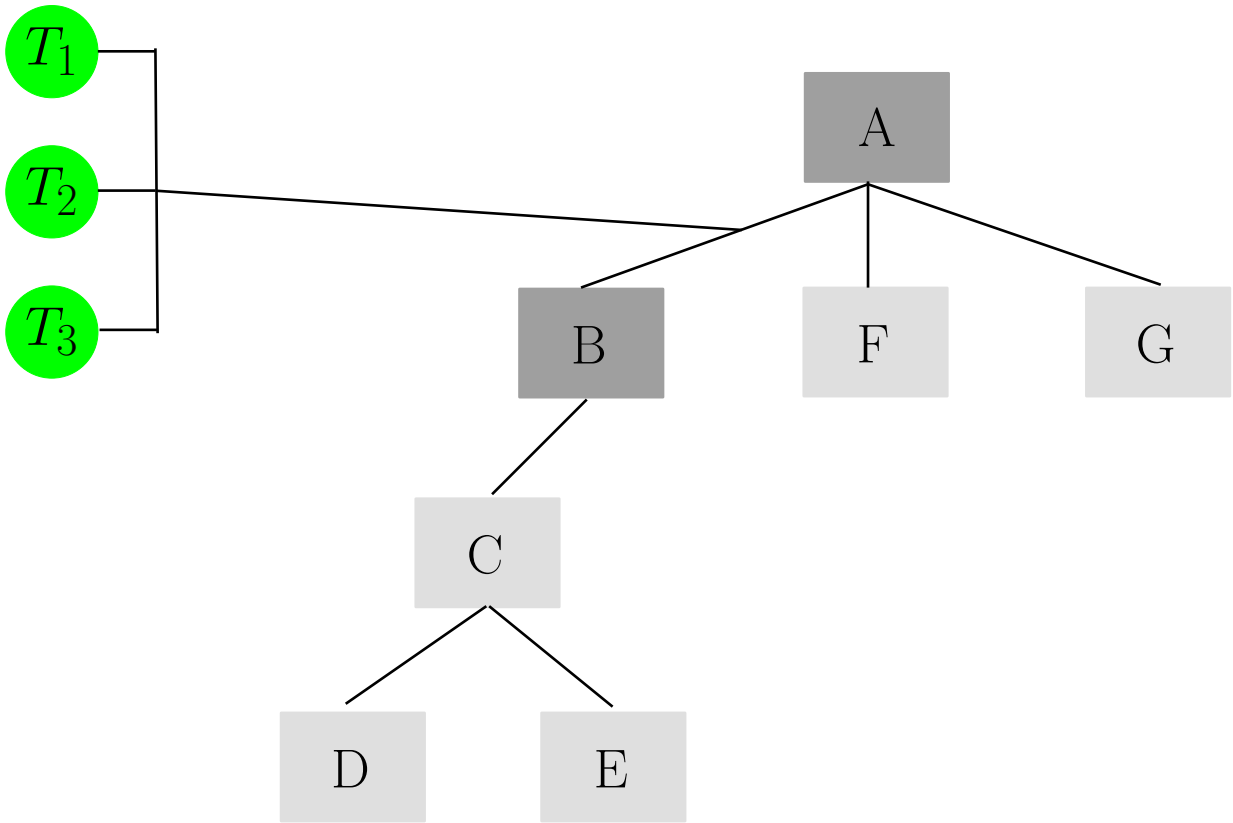
Pruebas de integración, descendente

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



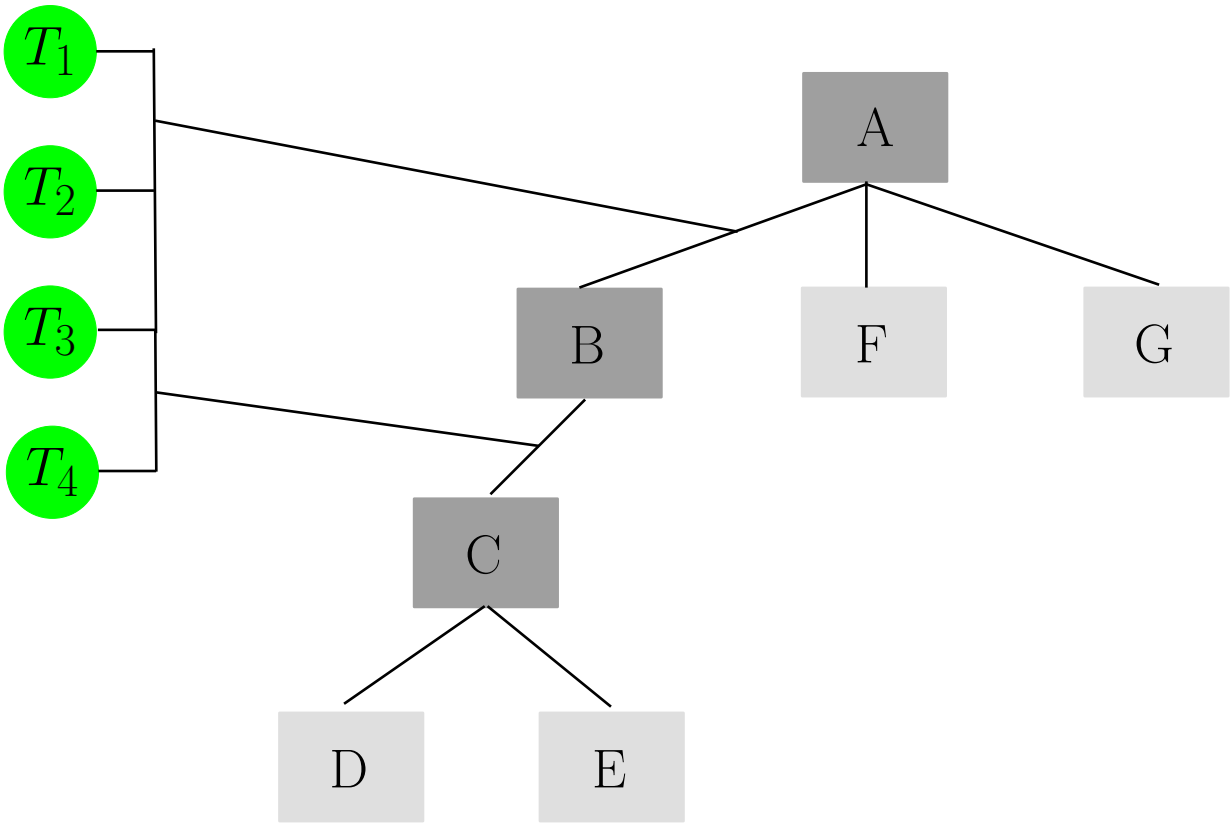
Pruebas de integración, descendente

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



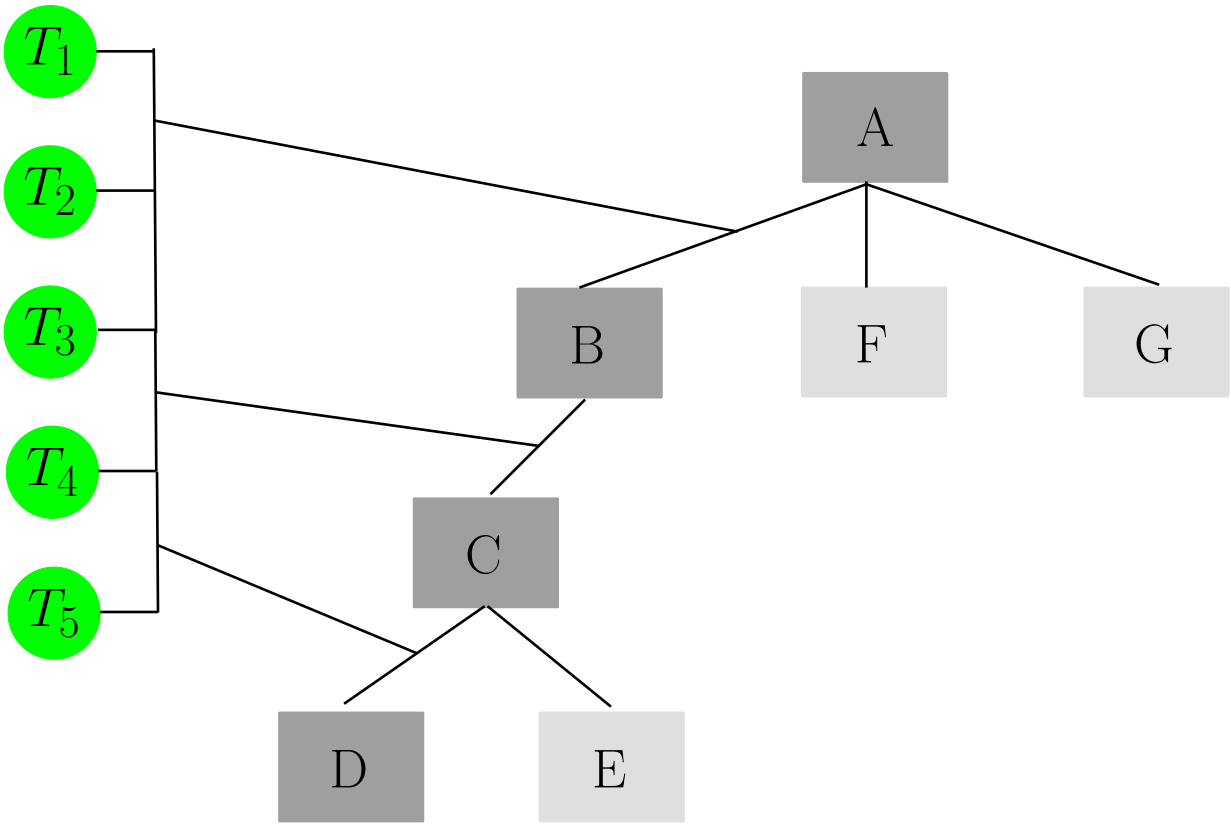
Pruebas de integración, descendente

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



Pruebas de integración, descendente

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

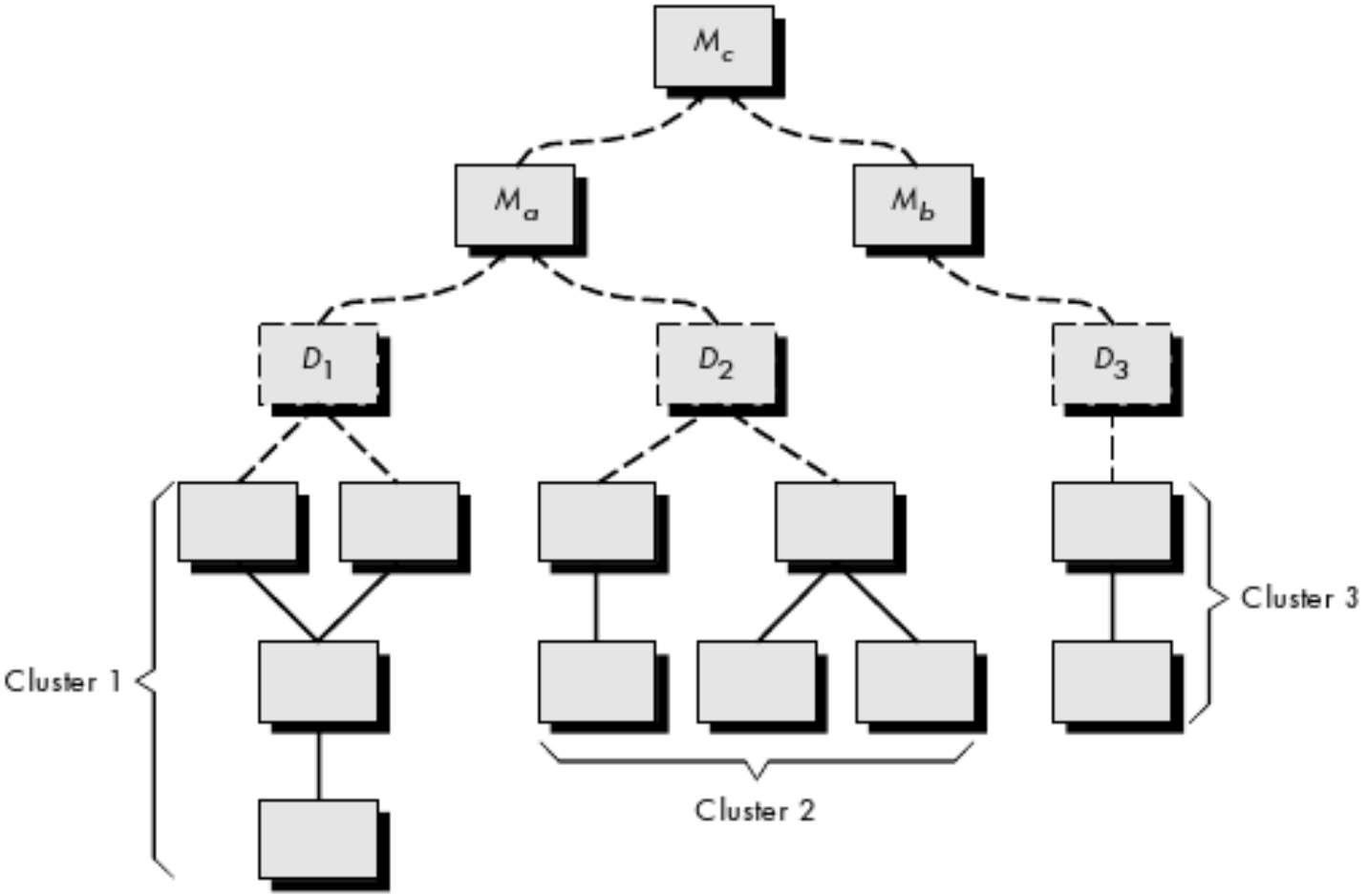


Pruebas de integración, ascendente

- Como su nombre lo indica, inicia la construcción y prueba con los módulos en los niveles más bajos de la estructura del programa
- No se requiere el uso de módulos simulados (*stubs*)
- La estrategia ascendente de integración puede ser implementada con los siguientes pasos:

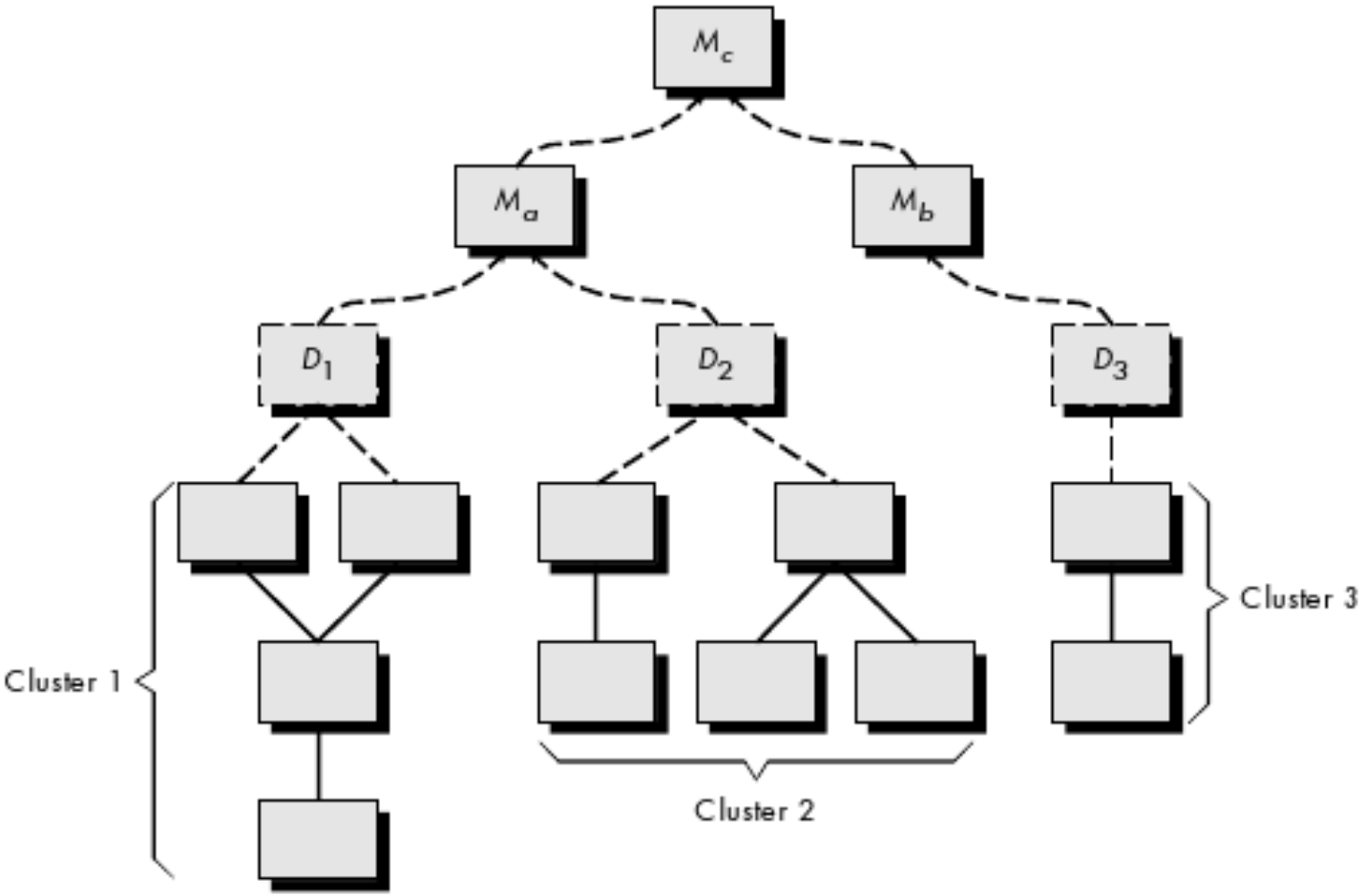
Pruebas de integración, ascendente

- Los módulos en niveles bajos de la jerarquía son combinados en grupos que realizan una subfunción específica del software



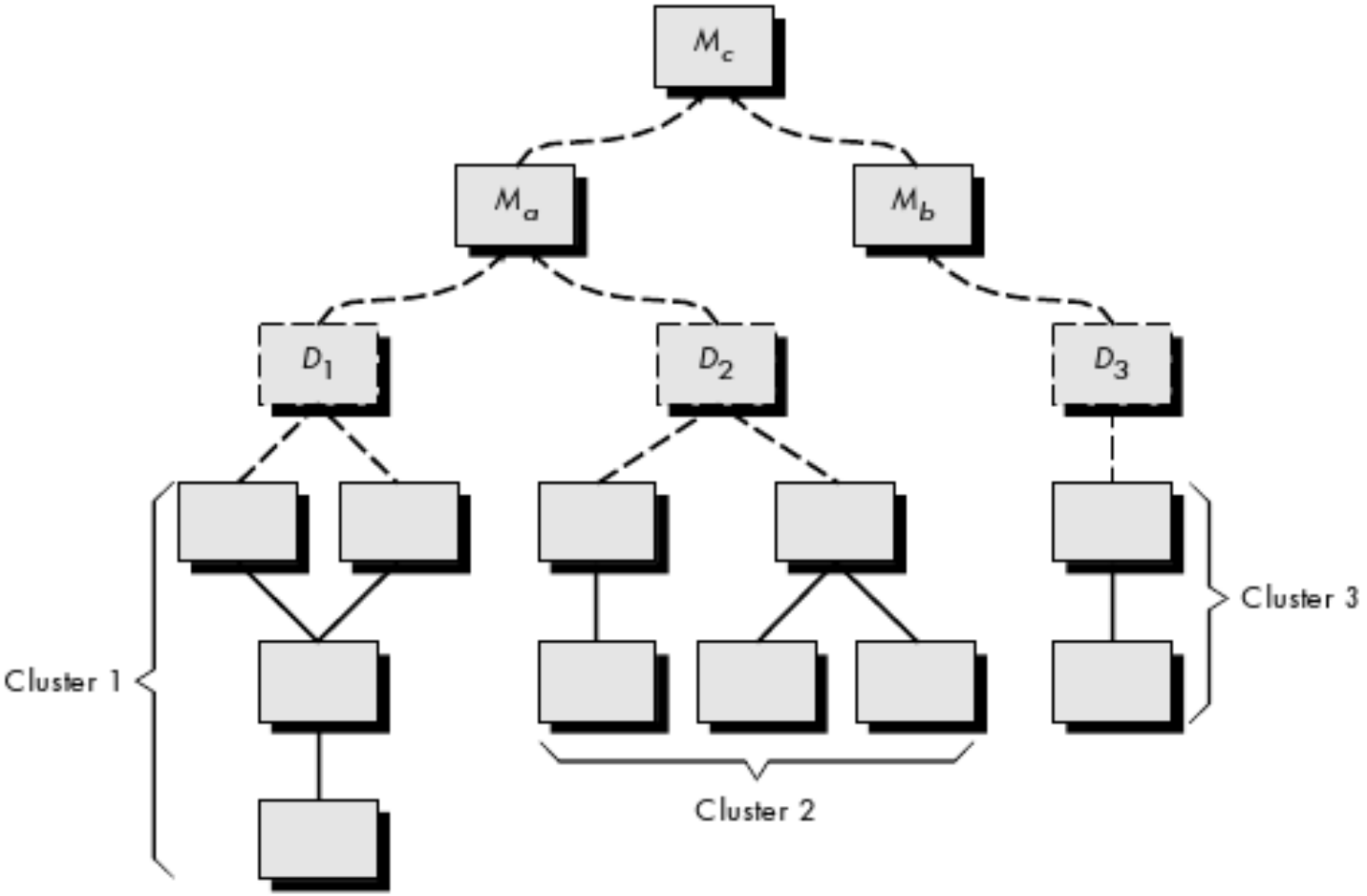
Pruebas de integración, ascendente

- Se escribe un módulo controlador para coordinar las entradas y salidas de los casos de prueba



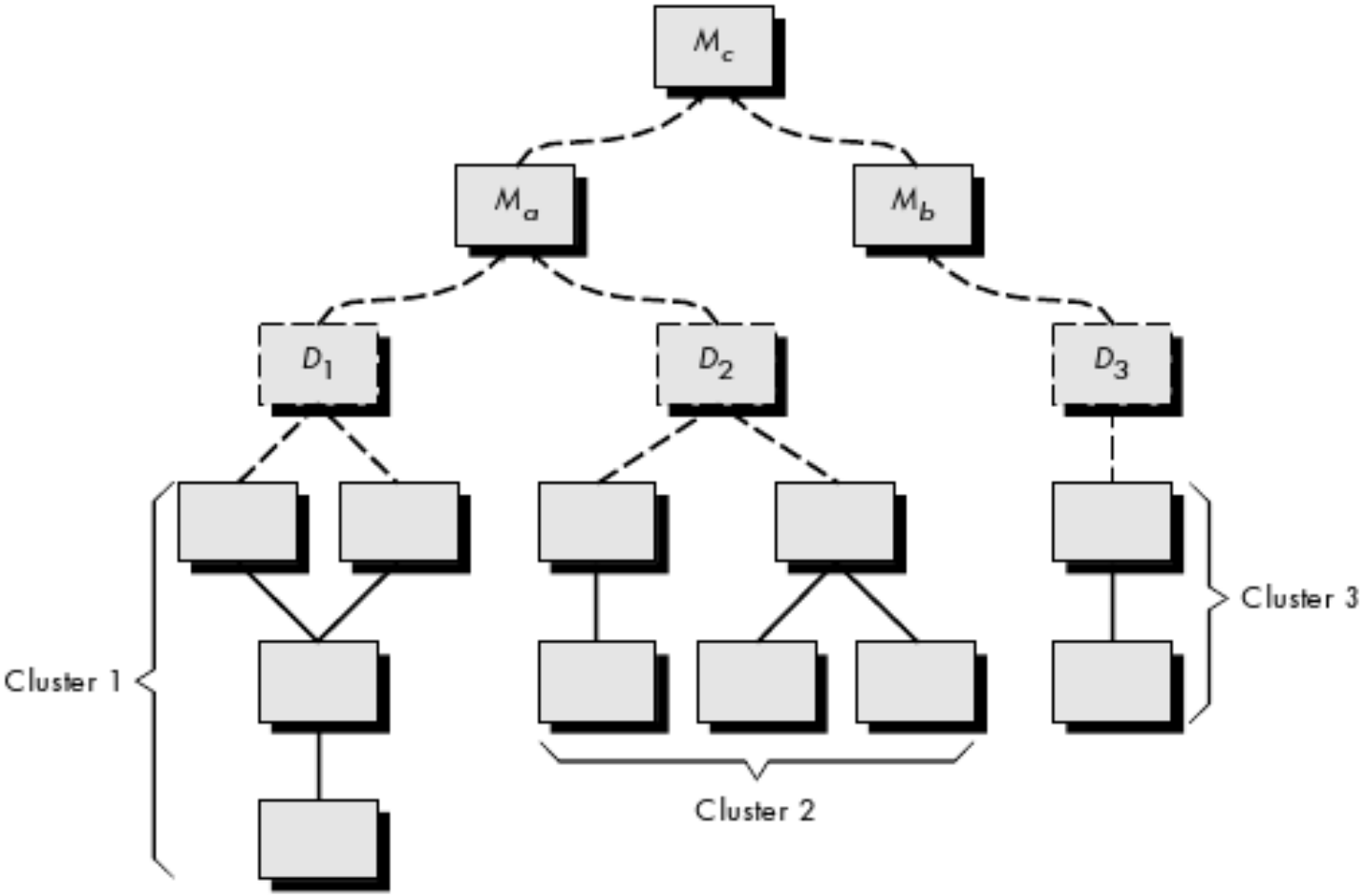
Pruebas de integración, ascendente

- El grupo es probado



Pruebas de integración, ascendente

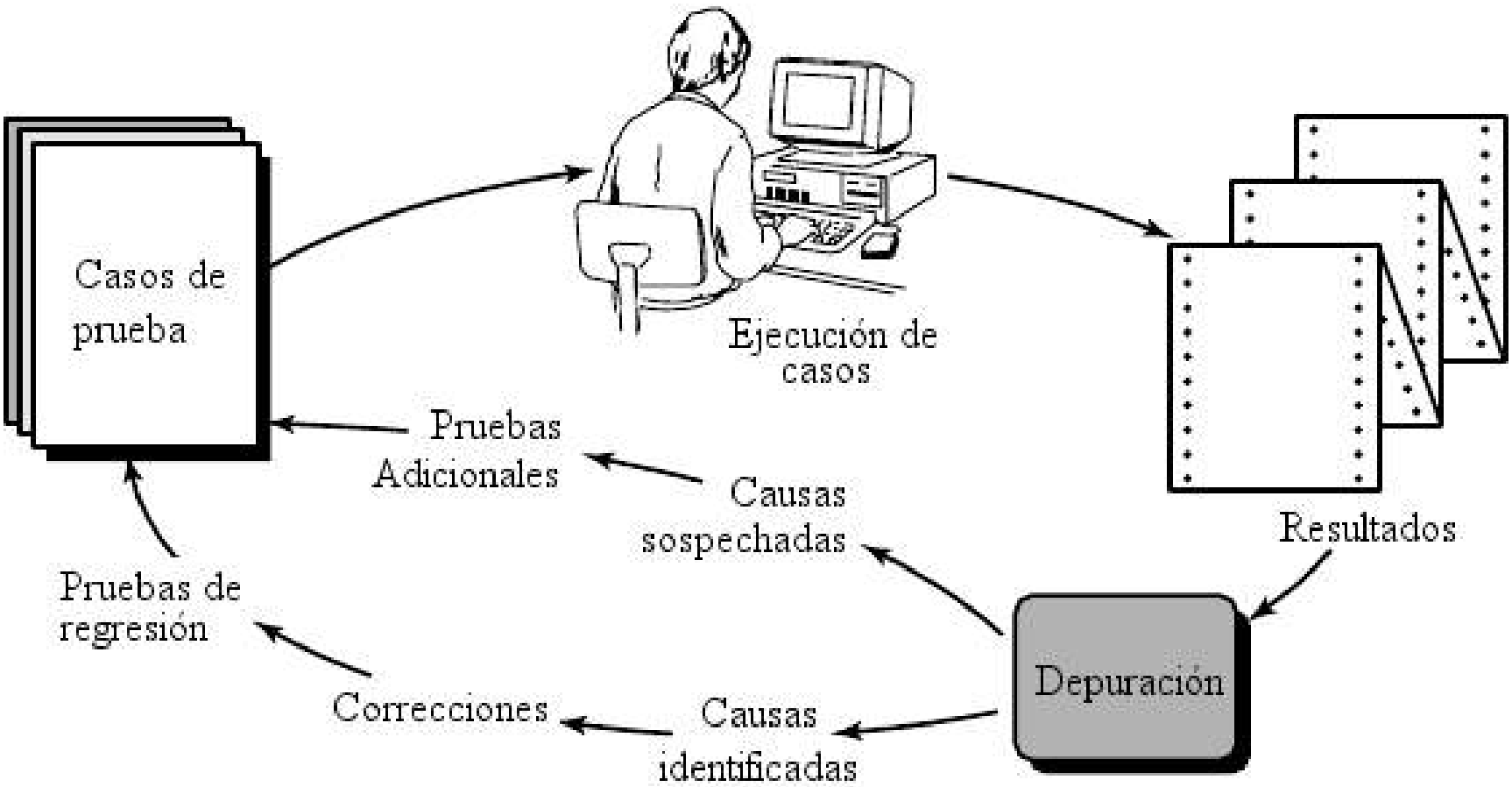
- Los controladores son remplazados y los grupos son combinados hacia arriba en la estructura del programa



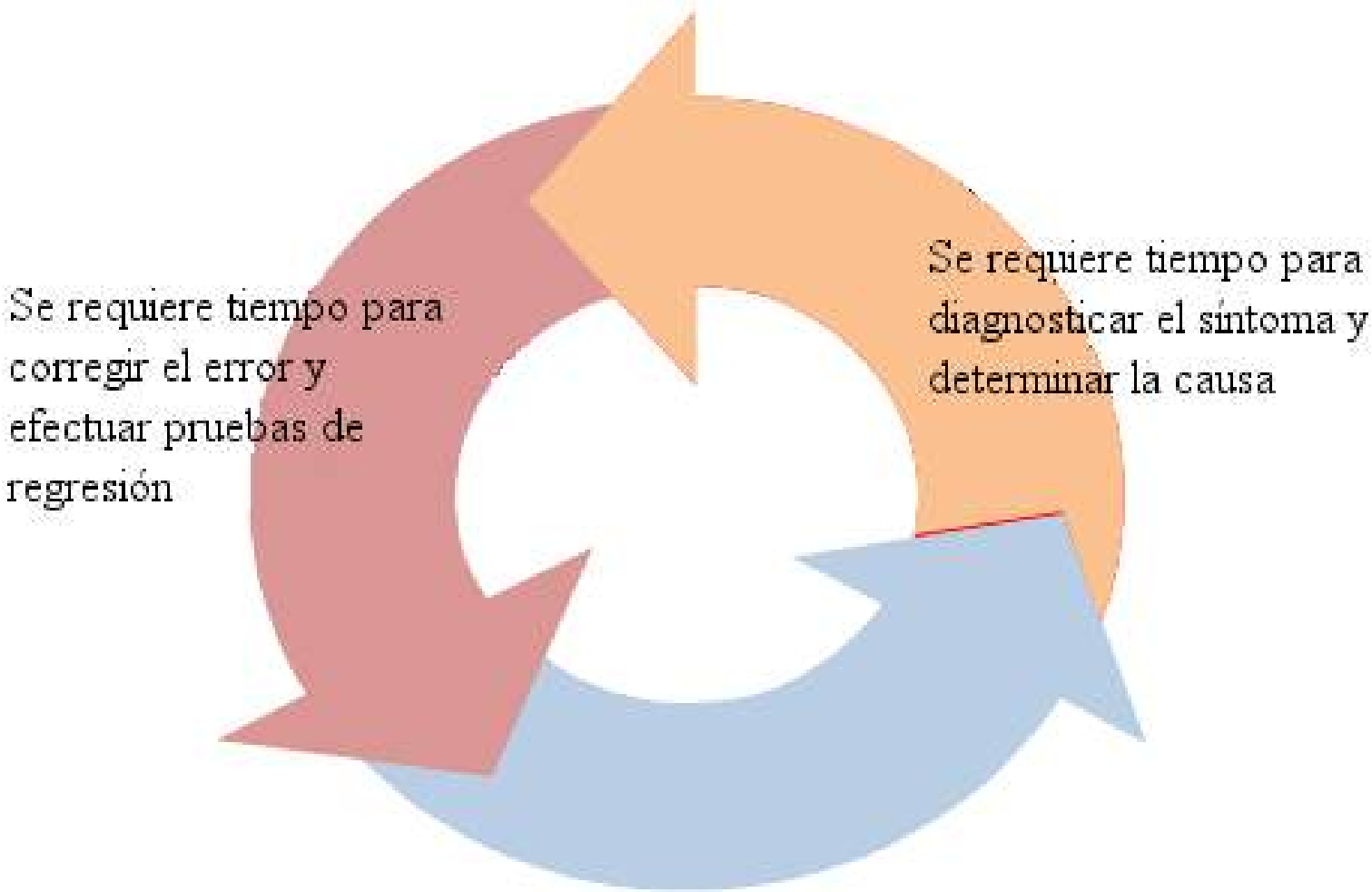
Pruebas de alto nivel

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

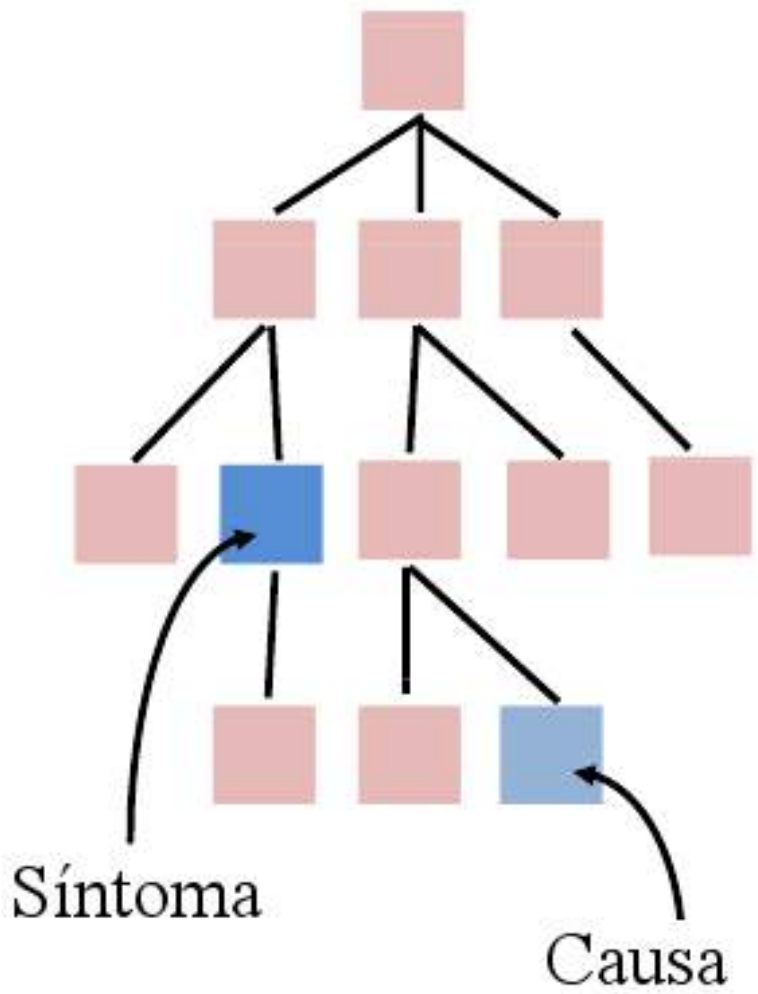
Depuración
Proceso de depuración



Depuración
Esfuerzo de depuración



Depuración
Síntomas y causas



- Síntoma y causa pueden estar separados
- Síntoma puede desaparecer al arreglar otro problema
- Causa puede darse por combinación de no-errores
- Causa puede darse por error de sistema o compilador
- Síntoma puede ser intermitente

Depuración
Herramientas de depuración en IDEs

- Existen en el mercado diversas herramientas de depuración
- En la mayoría de las IDEs se incluye alguna
- Veamos un ejemplo con Java y Netbeans

Depuración
Herramientas de depuración, Valgrind

- Valgrind es una herramienta que permite detectar fácilmente errores (bugs) en el manejo de memoria e hilos (threads)
- Funciona con programas escritos en cualquier lenguaje o mezclas de ellos gracias a que trabaja directamente con el código binario

Depuración
Herramientas de depuración, Valgrind

- Valgrind se ha empleado para detectar errores de memoria en numerosos proyectos importantes:
  - OpenOffice
  - Mozilla Firefox
  - MySQL
  - Perl
  - PHP
  - Yahoo! Messenger
  - PACX-MPI, etc.

Depuración
Herramientas de depuración, Valgrind

- Su nombre deriva de la mitología nórdica, donde Valgrind es la puerta principal del paraíso (Valhalla) al cual los héroes van al morir en combate
- Para instalarlo en Linux sólo se necesita ejecutar el comando siguiente en una consola

  sudo apt-get install valgrind

Depuración
Herramientas de depuración, Valgrind

- Para utilizar Valgrind es necesario seguir los siguientes pasos:
  1. Compilar tu programa con las opciones -O0 (O mayúscula cero) y -g
  2. Ejecutar el comando siguiente:

     valgrind –leak-check=yes tuPrograma p1 p2 p3

- Donde *tuPrograma* indica el nombre del archivo binario generado al compilar tu código y *p1, p2, p3* los parámetros que recibe de la línea de comandos tu programa

- Suponga el siguiente programa en C que se encuentra en un archivo llamado demos01.c

```
#include <stdio.h>
#include <stdlib.h>
void invalidReadAndWrite(void) {
  int* x = malloc(10 * sizeof (int));
  x[10] = 10; // problem: invalid write
  int y = x[11]; // problem: invalid read
} // problem: memory leak -- x not freed

void UseUninitialisedValues(void) {
  int j;
  if (j == 77)
   printf("hello there\n"); // problem: use of uninitialised
values
}
```

```
void illegalFrees(void) {
  int i, *p = malloc(10 * sizeof (int));
  for (i = 0; i < 10; i++)
    p[i] = i;
  free(p);
  free(p); // problem: p has already been freed
}

int main(int argc, char** argv) {
  invalidReadAndWrite();
  UseUninitialisedValues();
  illegalFrees();
  return 0;
}
```

Depuración
Herramientas de depuración, Valgrind

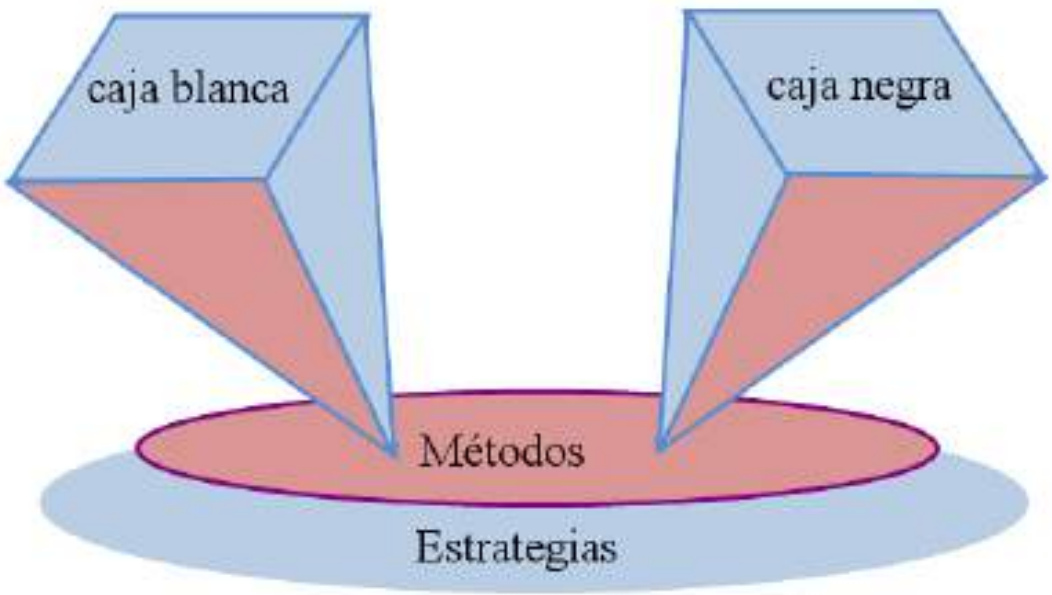
- Veamos que errores es capaz de detectar Valgrind en ese código...

Depuración
Herramientas de depuración, Valgrind

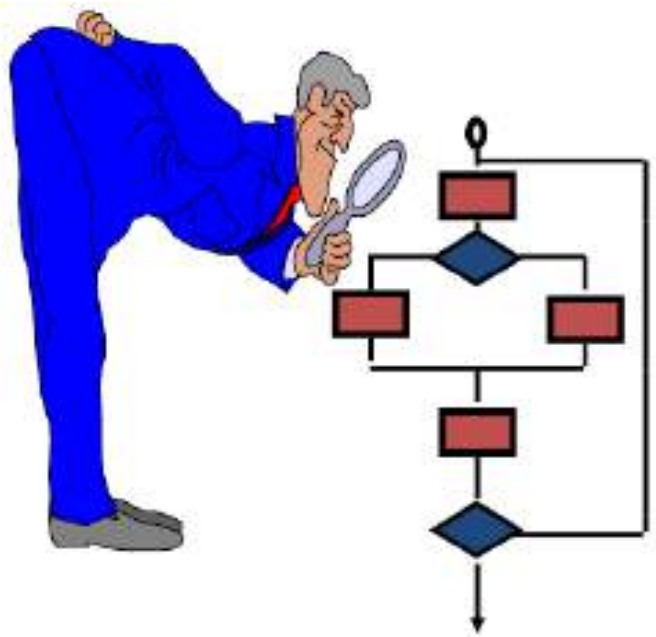
- Como pudimos observar Valgrind es muy útil para detectar los siguientes tipos de errores en el uso de memoria
  - Escritura en memoria no inicializada
  - Lectura de memoria fuera de rango
  - Condicionantes que dependen de memoria no inicializada
  - Incorrecta liberación de memoria (fugas)

Métodos de prueba

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

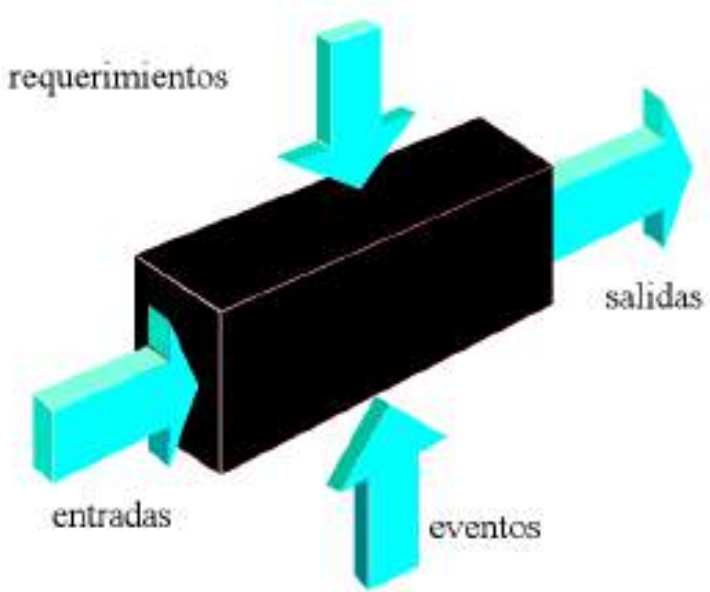


Métodos de prueba, de caja blanca



- 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 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

Pruebas 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

¿Por qué?

- Los errores lógicos y suposiciones incorrectas son inversamente proporcionales a la probabilidad de una ruta de ejecución
- A menudo se *cree* que una ruta no es probable que se ejecute; de hecho, la realidad es a menudo contra intuitiva
- Los errores tipográficos son aleatorios; es probable que las rutas no probadas contengan algunos errores de este tipo

Criterios de suficiencia de las pruebas

- En las pruebas de caja blanca se requiere poder decidir en qué elementos estructurales se enfocaran las pruebas, esto permite:
  - Elegir los datos de prueba adecuados
  - Decidir cuando los esfuerzo dedicados a las pruebas son suficientes para terminar el proceso
- Esto se conoce como *criterios de suficiencia de las pruebas*

Criterios de suficiencia de las pruebas

- Los *criterios de suficiencia de las pruebas* más comunes son:
  - Los basados en las propiedades estructurales de los programas (estructuras lógicas y de control, flujo de los datos, etc)
  - Los basados en las especificaciones de los programas
- Algunos incluso ignoran tanto la estructura como las especificaciones del programa, un ejemplo es el criterio aleatorio

Criterios de suficiencia de las pruebas

- A continuación veremos diferentes tipos de pruebas que se emplean principalmente bajo el criterio de suficiencia basado en las propiedades estructurales de los programas

Prueba de ruta básica

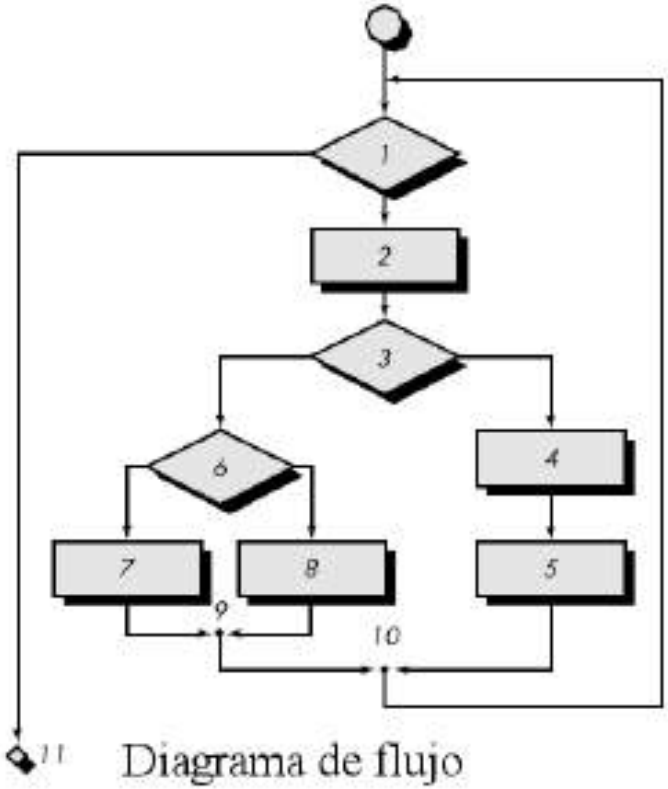
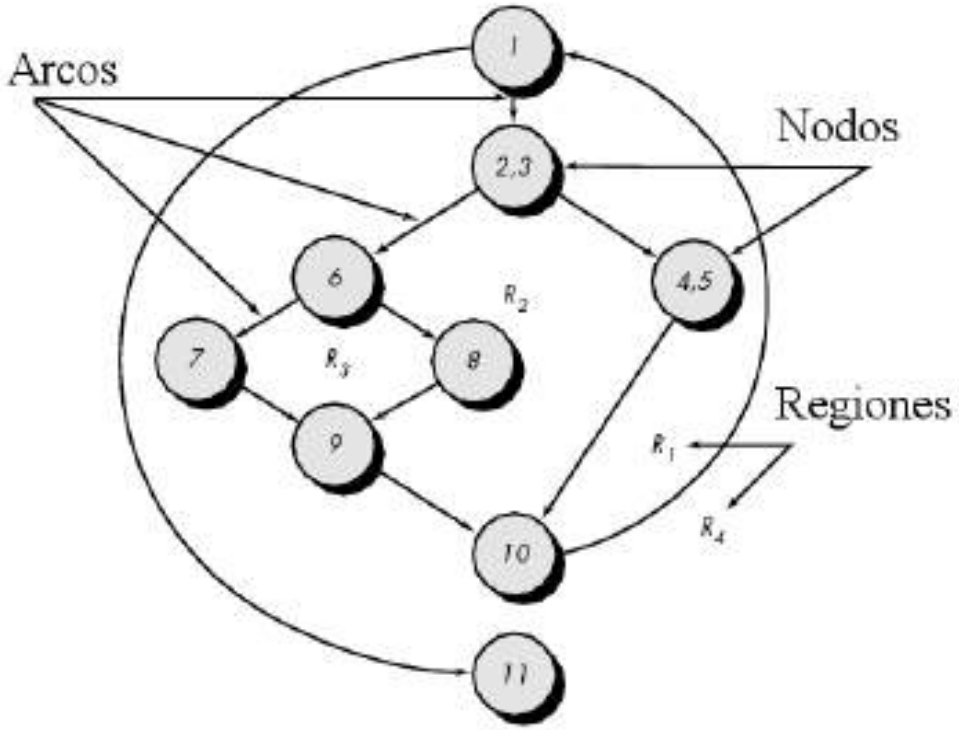


Diagrama de flujo



Grafo de flujo

- Cálculo de la **complejidad ciclomática**
- $V(G) = R = 4$
- $V(G) = E - V + 2 = 11 - 9 + 2 = 4$

Prueba de ruta básica

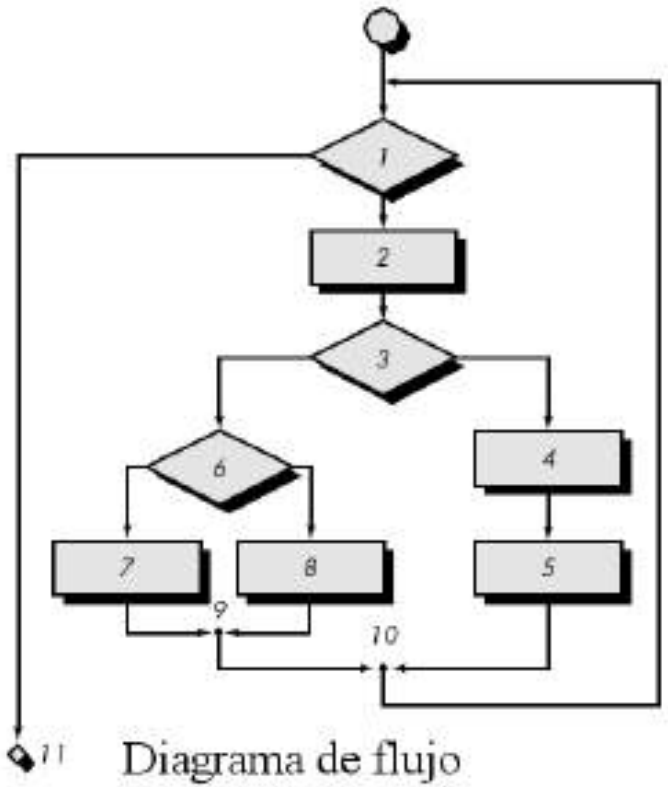


Diagrama de flujo

- Se derivan las rutas independientes ($V(G) = 4$)
- 1, 11
- 1, 2, 3, 4, 5, 10, 1, 11
- 1, 2, 3, 6, 8, 9, 10, 1, 11
- 1, 2, 3, 6, 7, 9, 10, 1, 11
- Se derivan casos de prueba para ejercitar cada ruta

Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```

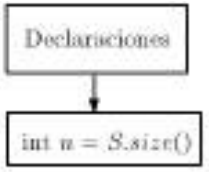
Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```

Declaraciones

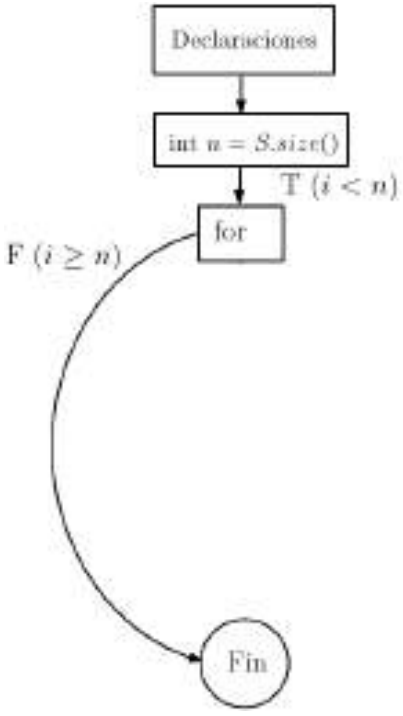
Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```



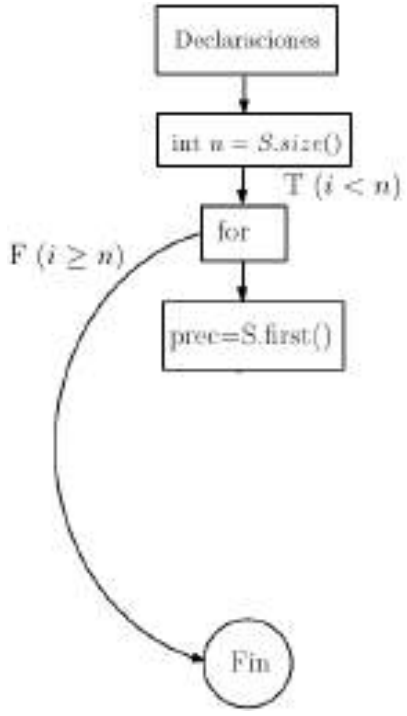
Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```



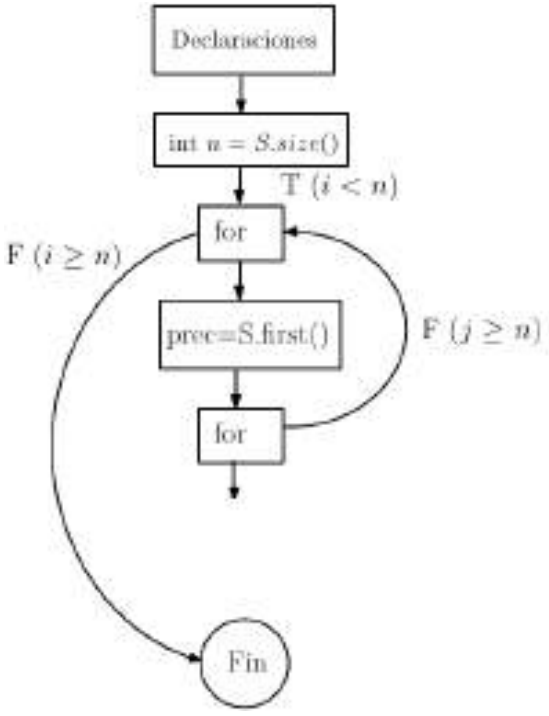
Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```



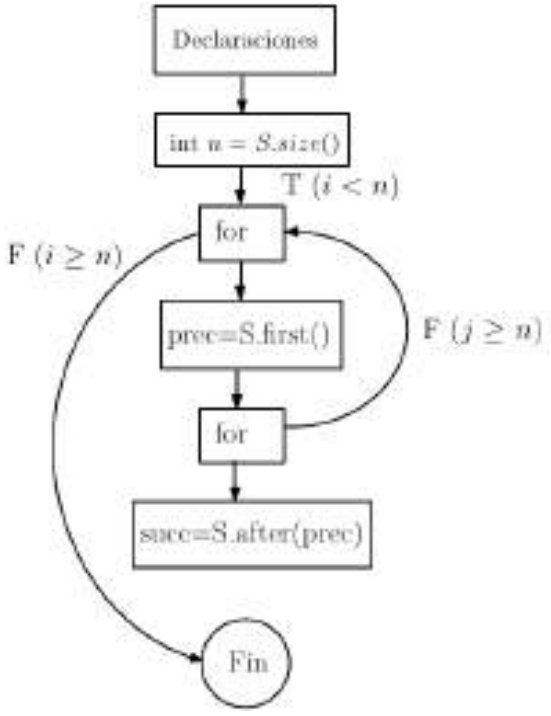
Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```



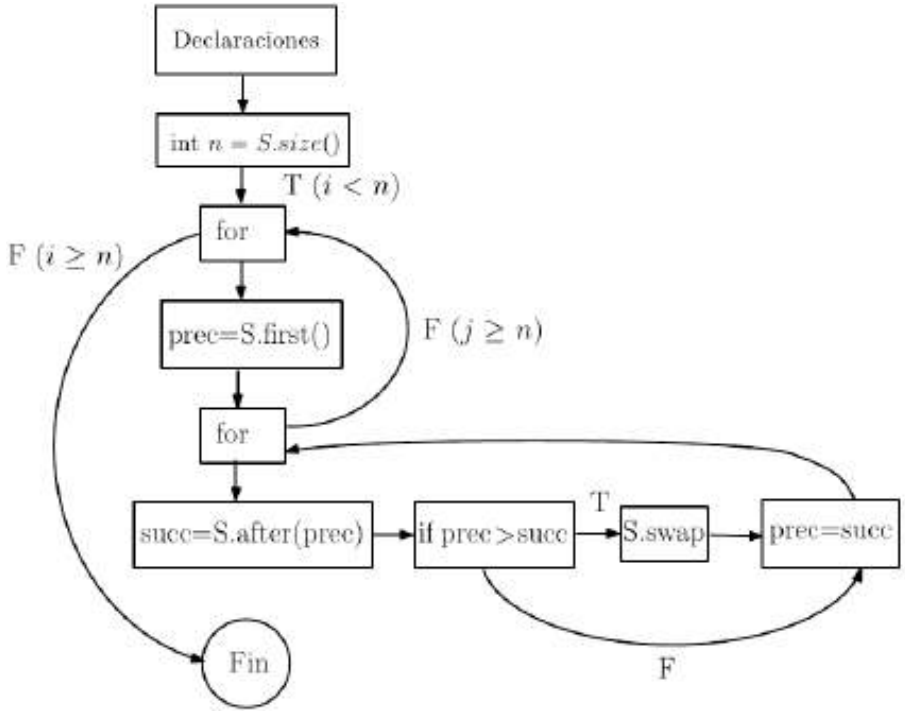
Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```

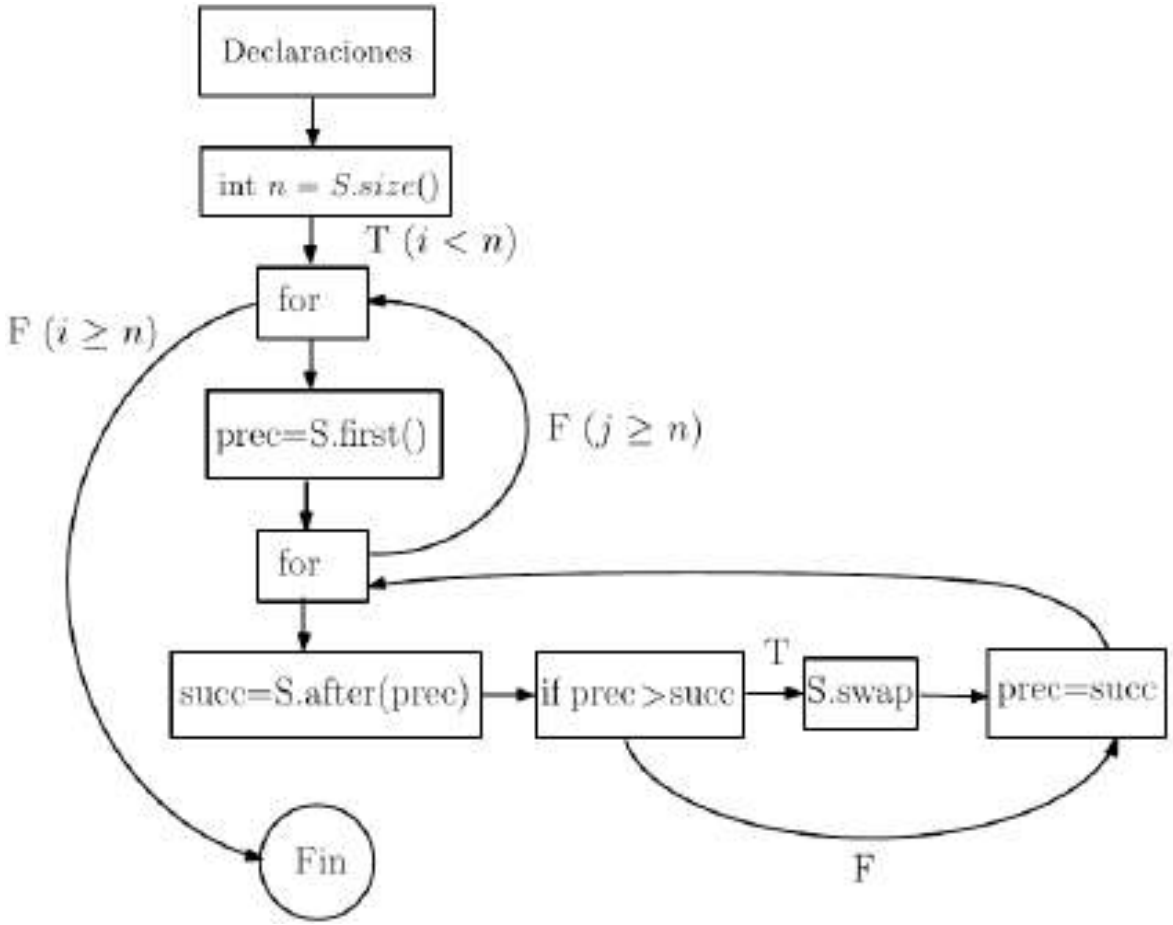


Prueba de ruta básica, Ejemplo

```
public static void bubbleSort2 (Sequence S) {
    Position prec, succ;
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        prec = S.first();
        for (int j=1; j < n - i; j++) {
            succ = S.after(prec);
            if (valAtPos(prec) > valAtPos(succ))
                S.swapElements(prec, succ);
            rec = succ;
        }
    }
}
```

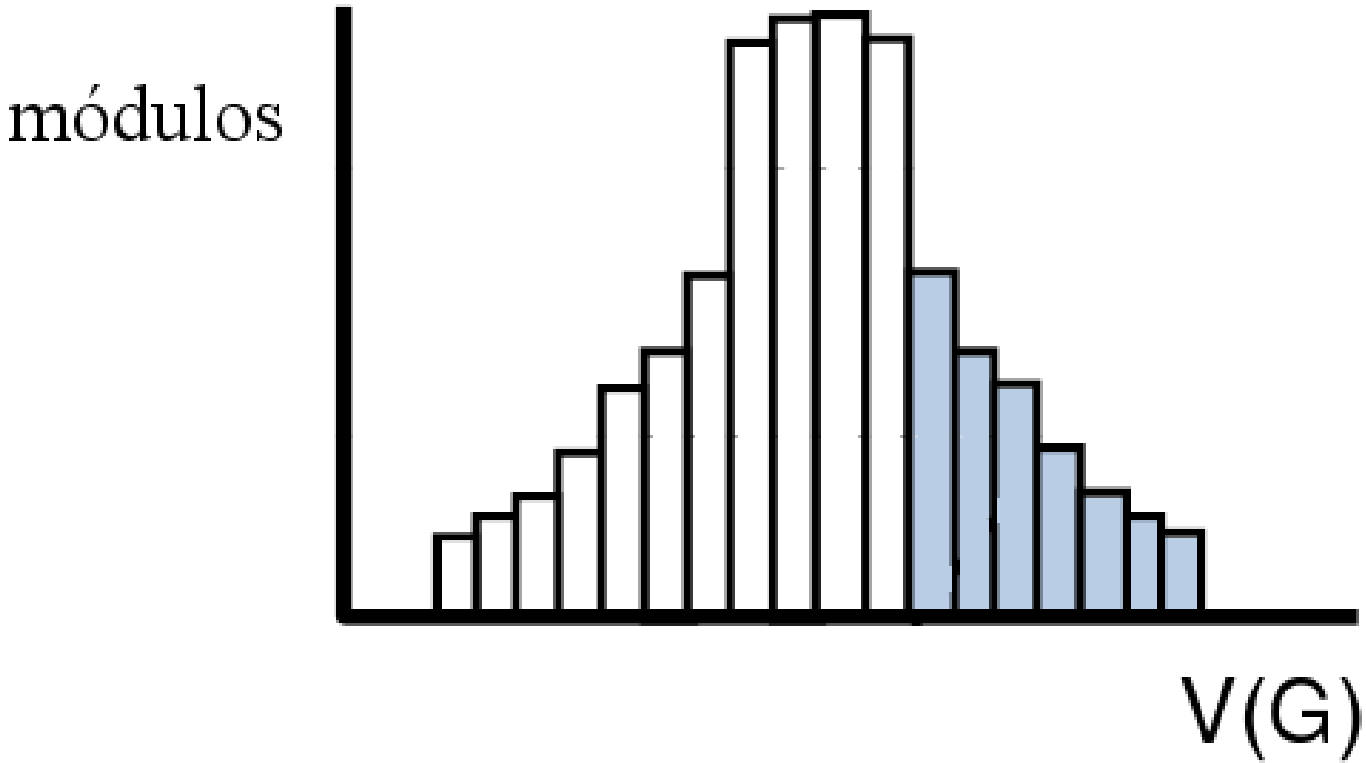


Prueba de ruta básica



- Complejidad ciclomática $V(G) = E - V + 2 = 12 - 10 + 2 = 4$

Prueba de ruta básica



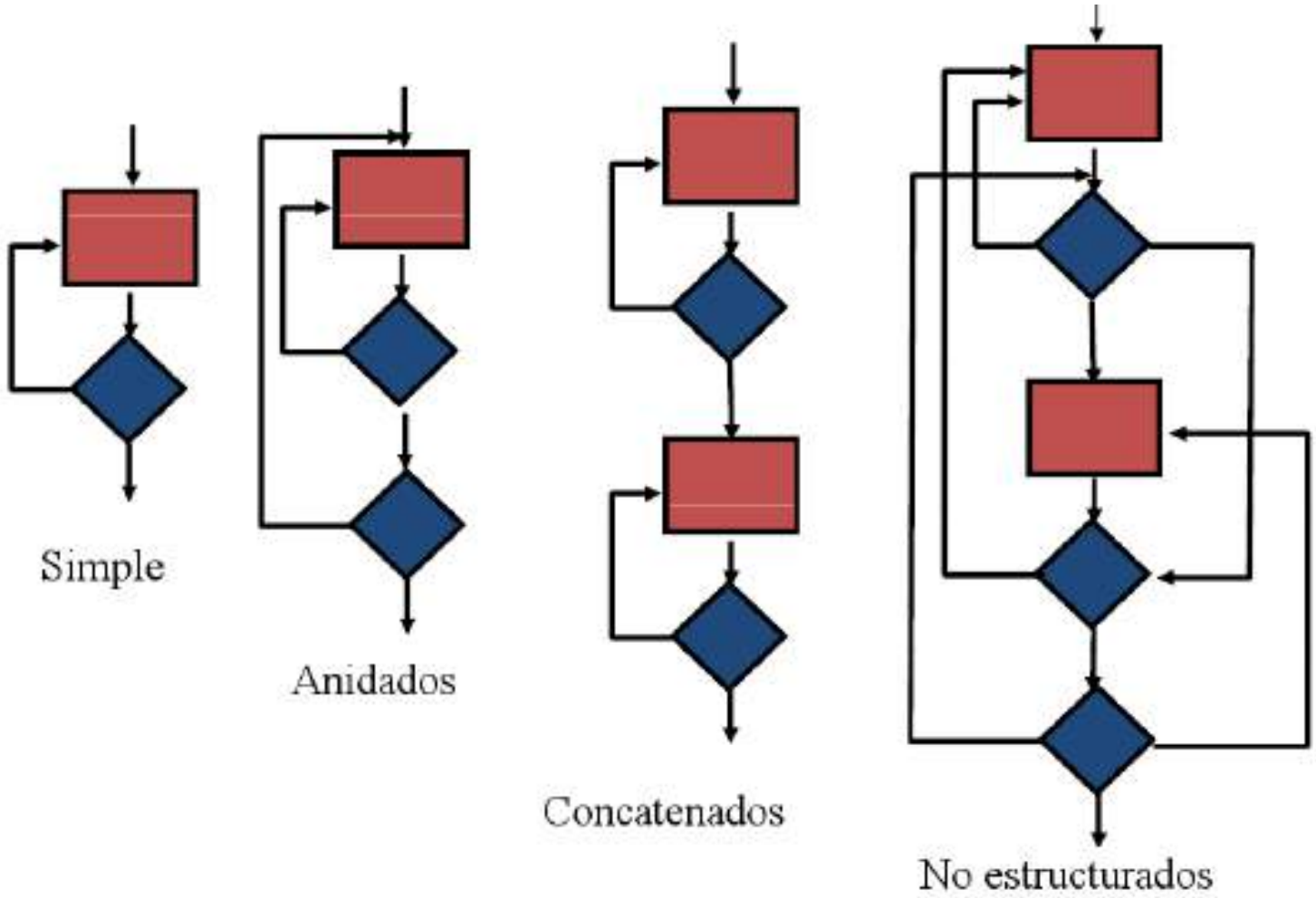
- Los módulos en este rango de complejidad ciclomática son más propensos a tener fallas

Prueba de la estructura de control

- **Prueba de condición**, un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en un módulo
- **Prueba de flujo de datos**, selecciona rutas de prueba del módulo de acuerdo con la localización de las definiciones y utilización de las variables del programa

Prueba de la estructura de control

Prueba de ciclos



Prueba de la estructura de control

Prueba de ciclos simples

- Condiciones mínimas para probar *ciclos simples*
  1. Saltar el ciclo completamente
  2. Sólo pasar una sola vez a través del ciclo
  3. Pasar dos veces a través del ciclo
  4. Pasar m veces a través del ciclo ($m < n$)
  5. Pasar $(n-1)$, n y $(n+1)$ veces a través del ciclo
- n es el número máximo de pasadas del ciclo

```
for (i=1; i < 100; i++)
{
...
}
```

Prueba de la estructura de control

Prueba de ciclos anidados

- Condiciones mínimas para probar *ciclos anidados*
  1. Iniciar en el ciclo más interno. Asignar a todos los ciclos sus valores de iteración mínimos
  2. Probar para este ciclo los valores $(min + 1)$, típico, $(max - 1)$ y max
  3. Pasar al siguiente ciclo más externo y fijar sus valores de iteración (paso 2), mantener los otros ciclos en sus valores típicos

```
for (i=1; i < 100; i++)
{
        for (j=1; j < 20; j++)
        {
                ...
        }
        ...
}
```

Prueba de flujo de datos

- Es un método de diseño de casos de prueba que se enfoca en el rol de las variables (datos) en el código
- Selecciona rutas de prueba del módulo de acuerdo con la localización de las definiciones y utilización de las variables

Prueba de flujo de datos

- Veamos algunos conceptos que definen el rol de las variables en un componente de software
- Por ejemplo, decimos que una variable está definida en una sentencia cuando su valor es asignado o cambiado

$$Y = 26 * X$$

- Y en este caso está definida. En la notación empleada en las pruebas de flujo de datos hablamos de un **def** para la variable Y

Prueba de flujo de datos

- Decimos que una variable es utilizada en una sentencia cuando su valor es empleado. El valor no cambia
- Existen dos tipos de utilización:
  - En un predicado, **p-use**

    if (X > 98)
  - En un cálculo, **c-use**

    *Y = 26 * X*

Prueba de flujo de datos

- Rapps y Weyuker proponen los siguientes criterios de suficiencia de las pruebas basadas en flujo de datos [Rapps and Weyuker, 1985]:
  - Todos los defs
  - Todos los p-use
  - Todos los c-use / algunos p-use
  - Todos los p-use / algunos c-use
  - Todos los use
  - Todos las rutas def-use (incluye todos los p- y c-use)

Prueba de flujo de datos, Ejemplo

1	sum = 0	*sum, def*
2	read (n),	*n, def*
3	i = 1	*i, def*
4	while (i <= n)	*i, n, p-use*
5	read (number)	*number, def*
6.	sum = sum + number	*sum, def, sum, number, c-use*
7	i = i + 1	*i, def, c-use*
8	end while	
9	print (sum)	*sum, c-use*

Variable	Id	def	use
n	1	2	4
number	1	5	6
sum	1	1	6
sum	2	1	9
sum	3	6	6
sum	4	6	9
i	1	3	4
i	2	3	7
i	3	7	7
i	4	7	4

Prueba de flujo de datos, Ejemplo

```
1       sum = 0
2       read (n),
3       i = 1
4       while (i <= n)
5               read (number)
6.              sum = sum + number
7               i = i + 1
8       end while
9       print (sum)
```

Variable	Id	def	use
n	1	2	4
number	1	5	6
sum	1	1	6
sum	2	1	9
sum	3	6	6
sum	4	6	9
i	1	3	4
i	2	3	7
i	3	7	7
i	4	7	4

1. Casos de prueba 1: $n = 0$
   - cubre par 1 para *n*
   - cubre par 2 para *sum*
   - cubre par 1 para *i*
2. Caso de prueba 2: $n = 5$, $number = 1, 2, 3, 4, 5$
   - cubre par 1 para *n*
   - cubre par 1 para *number*
   - cubre pares 1,3,4 para *sum*
   - cubre pares 1,2,3,4 para *i*

Prueba de flujo de datos, Ejemplo

- Observen que incluso para esta pequeña pieza de código se tienen
  - 4 tablas
  - 4 pares *def-use* para dos de las variables
- Las pruebas basadas en flujo de datos como con la mayoría de los métodos de caja blanca son más efectivas para realizar pruebas unitarias

Prueba de flujo de datos, Ejemplo

- Cuando el código se hace más complejo y hay más variables que considerar se vuelve mucho más tardado analizar sus roles en el flujo de datos, identificar rutas, y diseñar casos de prueba
- Otro problema con las pruebas basadas en flujo de datos ocurre en el caso de variables dinámicas como los punteros
- Finalmente no existen herramientas comerciales que provean soporte a este tipo de pruebas

Pruebas 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

Pruebas de caja negra
Prueba aleatoria

- Cada módulo de software tiene un dominio de entrada del cual los datos de prueba deben ser seleccionados
- Existen reportadas en la literatura diversas formas de seleccionarlos
- Si dichos datos de prueba se seleccionan de forma aleatoria entonces hablamos de *pruebas aleatorias*

Pruebas de caja negra
Prueba aleatoria

- Por ejemplo, si el dominio de entrada válido para un módulo son todos los enteros positivos entre 1 y 100, se podrían seleccionar aleatoriamente los valores 55, 24 y 3 para desarrollar tres casos de prueba
- Sin embargo, existen los siguientes puntos a considerar:
  - ¿Son los 3 valores adecuados para mostrar que el módulo cumple con su especificación?
  - ¿Existen otros valores diferentes que permitan revelar los defectos del módulo?
  - ¿Se deben usar valores fuera del dominio válido como datos de prueba?

Pruebas de caja negra
Prueba aleatoria

- El uso de pruebas aleatorias puede ahorrar mucho tiempo y esfuerzo en la selección de los datos de prueba
- Sin embargo, se debe tener en mente que esta forma de seleccionar los casos de prueba tiene muy poca posibilidad de producir un conjunto efectivo de datos de prueba
- Afortunadamente existen otros enfoques más estructurados que permiten afrontar las desventajas de las pruebas aleatorias
- Analizaremos algunos a continuación

Pruebas de caja negra
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)

Pruebas de caja negra
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 segundo tardariamos más de 11 días

Pruebas de caja negra
Partición equivalente

- Por lo tanto se han propuesto otros criterios para seleccionar casos de prueba
- Uno de ellos es la *partición equivalente* [Myers, 1979]
- Ésta divide en subconjuntos (clases de equivalencia) los valores de los parámetros del sistema (o módulo) al asumir que todos los elementos en el mismo subconjunto resultan en un comportamiento similar

Pruebas de caja negra
Partición equivalente

- Las pruebas por partición equivalente tienen las siguientes ventajas:
  - Elimina la necesidad de las pruebas exhaustivas
  - Es una guía para seleccionar el conjunto de datos de entrada para las pruebas, con alta probabilidad de detectar defectos
  - Permite realizar una cobertura de un dominio de entradas grande con un subconjunto pequeño tomado de cada clase de equivalencia

Pruebas de caja negra
Partición equivalente

- Algunas consideraciones deben ser tomadas en cuenta:
  1. Considerar clases de equivalencia tanto validas como invalidas. Las invalidas representan errores o entras invalidas
  2. Las clases de equivalencia deben también seleccionarse a partir de las condiciones de salida
  3. La derivación de las clases de equivalencia es un proceso heurístico. No existe una receta.
  4. En algunos casos es difícil identificar las clases de equivalencia debido a una mala especificación del módulo

Pruebas de caja negra
Partición equivalente, Ejemplo

```
Function square_root
message (x:real)
when x >= 0.0
reply (y:real)
where y >= 0.0 & approximately (y*y,x)
otherwise reply exception imaginary_square_root
end function
```

- Clases de equivalencia de entrada
  - EC1. La variable de entrada x es real y válida
  - EC2. La variable de entrada x no es real y inválida
  - EC3. El valor de $x > 0.0$ y es válido
  - EC4. El valor de $x < 0.0$ y es inválido

Pruebas de caja negra
Partición equivalente, Ejemplo

- Después de haber identificado las clases de equivalencia el siguiente paso es diseñar los casos de prueba
- Un buen enfoque incluye los siguientes pasos:
  1. Se debe asignar a cada clase de equivalencia un identificador único
  2. Desarrollar casos de prueba para las clases de equivalencia validas hasta que todas han sido cubiertas (un caso de prueba puede cubrir más de una clase)
  3. Desarrollar casos de prueba para las clases de equivalencia invalidas hasta que todas han sido cubiertas. Para garantizar que un caso de prueba inválido no oculte el efecto de otro (o impida ejecutarlo)

Pruebas de caja negra
Análisis de valores de frontera

- El *análisis de valores de frontera* requiere como su nombre lo dice que se seleccionen valores próximos a los límites de las clases de equivalencia
- Esto permite que los valores tanto en el límite inferior como superior de una clase de equivalencia sean cubiertos por los casos de prueba
- Al igual que las pruebas por partición equivalente la habilidad para desarrollar casos de prueba efectivos requiere de experiencia

Pruebas de caja negra

Partición equivalente y análisis de valores de frontera, ejemplo

- Suponga que deseamos probar un que permite al usuario capturar piezas para construir un auto en una base de datos
- Dichas piezas tienen un identificador que debe cumplir las siguientes condiciones:
  1. Consiste de caracteres alfanuméricos
  2. El número total de caracteres es entre 3 y 15
  3. Los primeros dos caracteres deben ser letras

Pruebas de caja negra

Partición equivalente y análisis de valores de frontera, ejemplo

Primero identificamos clases de equivalencia para la entrada

- Condición 1 (caracteres alfanuméricos)
  - EC1. El id de la parte es alfanumérico, válido
  - EC2. El id de la parte no es alfanumérico, inválido
- Condición 2 (número de caracteres entre 3 y 15)
  - EC3. El id de la parte tiene entre 3 y 15 caracteres, válido
  - EC4. El id de la parte tiene menos de 3 caracteres, inválido
  - EC5. El id de la parte tiene más de 15 caracteres, inválido
- Condición 3 (primeros 2 caracteres deben ser letras)
  - EC6. Los primeros 2 caracteres son letras, válido
  - EC7. Los primeros 2 caracteres no son letras, inválido

Pruebas de caja negra

Partición equivalente y análisis de valores de frontera, ejemplo

- Los resultados de la partición equivalente se refinan mediante un análisis de valores de frontera
- Para nuestro ejemplo los valores frontera son los siguientes:
  - Valor justo abajo del límite inferior (BLB) 2
  - Valor en el límite inferior (LB) 3
  - Valor justo arriba del límite inferior (ALB) 4
  - Valor justo abajo del límite superior (BUB) 14
  - Valor en el límite superior (UB) 15
  - Valor justo arriba del límite superior (AUB) 16

Pruebas de caja negra

Partición equivalente y análisis de valores de frontera, ejemplo

- El siguiente paso es seleccionar datos de entrada para los casos de prueba que cubran las clases de equivalencia y los valores frontera

Caso de prueba	Datos de entrada	Clases válidas límites cubiertos	Clases inválidas límites cubiertos
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc*	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6 (caso típico)	

Pruebas de caja negra

Estrategias combinatorias, pruebas de interacción

- Otra alternativa razonable son las estrategias de combinación (ver [Grindal et al., 2005])
- Son métodos de selección de casos de prueba que identifican parámetros de entrada individuales y los combinan basados en alguna estrategia combinatoria
- Un ejemplo de estas estrategias son las *pruebas de interacción* introducidas por Cohen et al. [Cohen et al., 1996]

Pruebas de caja negra

Estrategias combinatorias, pruebas de interacción

- 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 Arreglos Ortogonales (*Orthogonal Arrays*) son objetos combinatorios que pueden ser usados para representar esos casos de prueba de interacción [Hartman, 2005]
- Principalmente por que permiten maximizar el número de errores detectados reduciendo al mínimo el número de casos de prueba [Kuhn et al., 2004]

Pruebas de caja negra
Orthogonal Arrays

- Un *orthogonal array*, $\mathrm{OA}_\lambda(N; t, k, v)$ es un arreglo $N \times k$ sobre v símbolos tal que cada subarreglo $N \times t$ contiene todos los subconjuntos ordenados de tamaño t (t-tuplas) de v símbolos exactamente λ veces

Pruebas de caja negra
Orthogonal Arrays

- Un *orthogonal array*, $\mathrm{OA}_\lambda(N; t, k, v)$ es un arreglo $N \times k$ sobre v símbolos tal que cada subarreglo $N \times t$ contiene todos los subconjuntos ordenados de tamaño t (t-tuplas) de v símbolos exactamente λ veces
- $\mathrm{OA}_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

Pruebas de caja negra
Orthogonal Arrays

- Un *orthogonal array*, $\mathrm{OA}_\lambda(N; t, k, v)$ es un arreglo $N \times k$ sobre v símbolos tal que cada subarreglo $N \times t$ contiene todos los subconjuntos ordenados de tamaño t (t-tuplas) de v símbolos exactamente λ veces
- $\mathrm{OA}_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

Pruebas de caja negra
Orthogonal Arrays

- Un *orthogonal array*, $\mathrm{OA}_\lambda(N; t, k, v)$ es un arreglo $N \times k$ sobre v símbolos tal que cada subarreglo $N \times t$ contiene todos los subconjuntos ordenados de tamaño t (t-tuplas) de v símbolos exactamente λ veces
- $\mathrm{OA}_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

Pruebas de caja negra
Orthogonal Arrays

- El mínimo N para el cual un $\mathrm{OA}_\lambda(N; t, k, v)$ existe es conocido como el tamaño óptimo del orthogonal array y se define como sigue:

$$\mathrm{OAN}_\lambda(t, k, v) = min\{N : \exists\, \mathrm{OA}_\lambda(N; t, k, v)\}$$

- El mínimo N puede ser calculado con la siguiente relación:

$$\mathrm{OAN}_\lambda(t, k, v) = \lambda v^t$$

- El orthogonal array de nuestro ejemplo es óptimo: $\mathrm{OA}_1(9; 2, 4, 3)$ $(N = 3^2 = 9)$

Pruebas de caja negra
Orthogonal Arrays

- Veamos un ejemplo de cómo se puede utilizar un orthogonal array para representar casos de prueba de interacción
- Supongamos un sistema de software que tiene 4 parámetros cada uno de los cuales puede tomar 3 valores diferentes y que queremos realizar la prueba de todas las interacciones entre pares de parámetros

Pruebas de caja negra, Orthogonal Arrays

Valor	Hardware	Navegador	SO	Conexión
0	PC	IE	XP	LAN
1	Laptop	Mozilla	Win7	ISDN
2	Tableta	Chrome	Linux	WiFi

Pruebas de caja negra, Orthogonal Arrays

Valor	Hardware	Navegador	SO	Conexión
0	PC	IE	XP	LAN
1	Laptop	Mozilla	Win7	ISDN
2	Tableta	Chrome	Linux	WiFi

- Los casos de prueba de interacción para este software se pueden representar con un $OA_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

Pruebas de caja negra, Orthogonal Arrays

Valor	Hardware	Navegador	SO	Conexión
0	PC	IE	XP	LAN
1	Laptop	Mozilla	Win7	ISDN
2	Tableta	Chrome	Linux	WiFi

- Los casos de prueba de interacción para este software se pueden representar con un $OA_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix} \qquad \begin{pmatrix} PC \\ \\ \\ \\ \\ PC \\ PC \\ \\ \end{pmatrix}$$

Pruebas de caja negra, Orthogonal Arrays

Valor	Hardware	Navegador	SO	Conexión
0	PC	IE	XP	LAN
1	Laptop	Mozilla	Win7	ISDN
2	Tableta	Chrome	Linux	WiFi

- Los casos de prueba de interacción para este software se pueden representar con un $OA_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix} \qquad \begin{pmatrix} \\ PC \\ Laptop \\ \\ \\ \\ PC \\ PC \\ Laptop \\ Laptop \end{pmatrix}$$

Pruebas de caja negra, Orthogonal Arrays

Valor	Hardware	Navegador	SO	Conexión
0	PC	IE	XP	LAN
1	Laptop	Mozilla	Win7	ISDN
2	Tableta	Chrome	Linux	WiFi

- Los casos de prueba de interacción para este software se pueden representar con un $OA_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix} \quad \begin{pmatrix} \textit{Tableta} \\ \textit{PC} \\ \textit{Laptop} \\ \textit{Tableta} \\ \textit{Tableta} \\ \textit{PC} \\ \textit{PC} \\ \textit{Laptop} \\ \textit{Laptop} \end{pmatrix}$$

Pruebas de caja negra, Orthogonal Arrays

Valor	Hardware	Navegador	SO	Conexión
0	PC	IE	XP	LAN
1	Laptop	Mozilla	Win7	ISDN
2	Tableta	Chrome	Linux	WiFi

- Los casos de prueba de interacción para este software se pueden representar con un $OA_1(9; 2, 4, 3)$

$$\begin{pmatrix} 2 & 1 & 2 & 2 \\ 0 & 2 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

$$\begin{pmatrix} \text{Tableta} & \text{Mozilla} & \text{Linux} & \text{WiFi} \\ \text{PC} & \text{Chrome} & \text{Win7} & \text{WiFi} \\ \text{Laptop} & \text{Chrome} & \text{Linux} & \text{ISDN} \\ \text{Tableta} & \text{Chrome} & \text{XP} & \text{LAN} \\ \text{Tableta} & \text{IE} & \text{Win7} & \text{ISDN} \\ \text{PC} & \text{IE} & \text{Linux} & \text{LAN} \\ \text{PC} & \text{Mozilla} & \text{XP} & \text{ISDN} \\ \text{Laptop} & \text{Mozilla} & \text{Win7} & \text{LAN} \\ \text{Laptop} & \text{IE} & \text{XP} & \text{WiFi} \end{pmatrix}$$

Pruebas de caja negra
Orthogonal Arrays

- Los orthogonal arrays son muy útiles para representar casos de prueba de interacción
- Representan una buena alternativa frente a las pruebas exhaustivas
- En nuestro ejemplo el $OA_1(9; 2, 4, 3)$ contiene 9 casos de prueba contra $3^4 = 81$ que serían necesarios en una prueba exhaustiva

Pruebas de caja negra
Orthogonal Arrays

- Sin embargo, existen muchos valores de t, k y v para los cuales no existe un orthogonal array con $\lambda = 1$
- Pues es muy restrictivo el hecho de que deban aparecer una sola vez cada t-tupla de v símbolos
- La alternativa es utilizar otros objetos combinatorios llamados *covering arrays* que son más flexibles

Pruebas de caja negra
Covering arrays

- Un *covering array*, CA($N; t, k, v$), de tamaño N, fuerza t, grado k, y orden v es un arreglo $N \times k$ sobre v símbolos tal que para cada subarreglo $N \times t$ contiene todos los subconjuntos ordenados de v símbolos de tamaño t (t-tuplas) al menos una vez

Pruebas de caja negra

Covering arrays

- El mínimo N para el cual un CA$(N; t, k, v)$ existe es conocido como el tamaño óptimo del covering array y se define como sigue:

$$\text{CAN}(t, k, v) = min\{N : \exists \text{CA}(N; t, k, v)\}$$

- Un límite inferior para N es el siguiente:

$$v^t \leq \text{CAN}(t, k, v)$$

Pruebas de caja negra
Covering arrays

- $CA(10; 3, 5, 2)$

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Pruebas de caja negra
Covering arrays

- $CA(10; 3, 5, 2)$

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Pruebas de caja negra

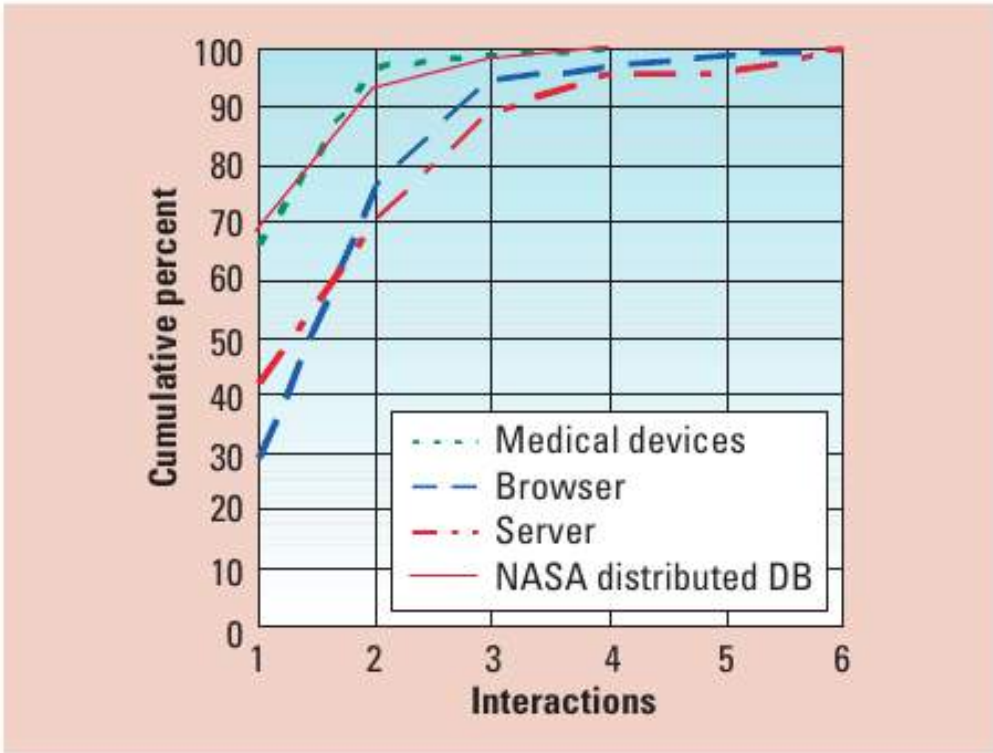
Covering arrays

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

Pruebas de caja negra
Covering arrays

- Algunos estudios realizados [Kuhn et al., 2004, Kuhn et al., 2008] han demostrado que el uso de covering arrays permiten reducir el número de casos de prueba y maximizar la posibilidad de detectar defectos



Pruebas de caja negra
Covering arrays

- El problema de construcción de covering arrays (CAC) consiste en determinar el tamaño óptimo (N) de un covering array dados sus valores t, k y v
- Es equivalente al problema de maximizar el grado k dados los valores N, t y v
- En general determinar el tamaño óptimo de un covering array es un problema NP-complete [Lei and Tai, 1998]
- Además es un problema altamente combinatorio, veamos por qué

Pruebas de caja negra
Covering arrays

- Sea A una solución potencial en el *espacio de búsqueda* $\mathscr{A}$, i.e., un covering array CA$(N; t, k, v)$
- Entonces A puede ser representada como un arreglo $N \times k$ sobre v símbolos

0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0
1	0	1	1

- CA$(9; 3, 4, 2)$
- $N = 9$
- $t = 3$
- $k = 4$
- $v = 2$

- El tamaño del espacio de búsqueda $\mathscr{A}$ es entonces dado por la expresión: $|\mathscr{A}| = v^{Nk}$ (e.g. $2^{9*4} = 68,719,476,736$)

Pruebas de caja negra
Covering arrays

- Tan solo evaluar si una solución potencial A es un covering array requiere verificar cuantas t-tuplas no están cubiertas:

$$\mathscr{F}(A) = \binom{k}{t} v^t - \sum_{j=0}^{\binom{k}{t}-1} |\vartheta_j|$$

- donde ϑ_j es un conjunto que contiene la unión de las N t-tuplas en el j-ésimo subconjunto de t columnas tomadas de k (un subarreglo $N \times t$)

- Complejidad computacional $O(N\binom{k}{t}) = O(\frac{Nk!}{t!(k-t)!})$

Pruebas de caja negra
Covering arrays

- A pesar de estas circunstancias, diferentes enfoques de solución han sido reportados en la literatura
- Algoritmos exactos:
  - EXACT (Exhaustive search) [Yan and Zhang, 2008]
  - Branch & Bound [Bracho-Rios et al., 2009]

Pruebas de caja negra
Covering arrays

- Algoritmos aproximados:
  - Métodos recursivos [Sloane, 1993]
  - Métodos algebraicos [Chateauneuf and Kreher, 2002]
  - Métodos greedy [Cohen et al., 1997]
  - Metaheurísticas
    - Tabu Search [Nurmela, 2004]
    - Simulated Annealing [Torres-Jimenez and Rodriguez-Tello, 2012]
    - Genetic Algorithms [Stardom, 2001]

Pruebas de caja negra
Covering arrays

- Veamos algunos ejemplos...

Pruebas de caja negra
Mixed level covering arrays

- En la practica, los módulos de software que se desean probar no siempre tienen el mismo número de valores (v) para todos sus parámetros
- Por lo que los covering arrays no pueden ser empleados para construir casos de prueba
- Para superar esta limitación se emplean otros objetos combinatorios conocidos como *mixed level covering arrays* (MCAs)

Pruebas de caja negra

Mixed level covering arrays

- Un mixed level covering array MCA($N; t, k, (v_1, v_2, \cdots, v_k)$) es un arreglo $N \times k$ sobre v símbolos ($v = \sum_{i=1}^{k} v_i$), donde cada columna i ($1 \leq i \leq k$) de este arreglo contiene solamente elementos de un conjunto S_i, con $|S_i| = v_i$.

- Este arreglo tiene la propiedad de que las filas de cada subarreglo $N \times t$ cubre las t-tuplas de valores de las t columnas al menos una vez

Pruebas de caja negra
Mixed level covering arrays

- Suponga que tenemos un algoritmo (Simulated Annealing) que tiene $k = 5$ parámetros cuyos valores se describen en la siguiente tabla y que deseamos probar todas las interacciones entre $t = 4$ parámetros

IS	α	t_i	$\mathcal{N}$	*CL*
Greedy	0.99	6.0	SPR	$15(n+k)$
Random	0.85	$(k+n)^{(1,0/3,3)}$	TBR	$23(n+k)$
-	-	-	-	$40(n+k)$

Pruebas de caja negra
Mixed level covering arrays

- Podríamos construir el mixed level covering array más pequeño: MCA$(24; 4, 5, (2, 2, 2, 2, 3))$

1	0	1	1	1	0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	1	0	1	1
1	0	0	0	1	0	0	1	0	0	1	0	1	0	1	0	0	1	0	1	1	1	1	1
1	1	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	1	0	1
1	2	1	1	1	0	0	0	2	1	0	0	2	2	2	0	2	1	1	1	2	0	0	2

- Y ejecutar sólo 24 casos de prueba en vez de $(2^4)3 = 48$ en una prueba exhaustiva

Pruebas de caja negra
Mixed level covering arrays

- Tanto los CA como los MCA tiene otras aplicaciones prácticas [Chateauneuf and Kreher, 2002]:
  - Desarrollo de fármacos
  - Compresión de datos
  - Estudio de la regulación de expresión génica
  - Autenticación, etc.

Estrategias y técnicas de prueba del software

Dr. Eduardo A. Rodríguez Tello

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

24 de octubre del 2012

Referencias I

Bracho-Rios, J., Torres-Jimenez, J., and Rodriguez-Tello, E. (2009).
A new backtracking algorithm for constructing binary covering arrays of variable strength.
Lecture Notes in Artificial Intelligence, 5845:397–407.

Burnstein, I. (2003).
Practical Software Testing.
Springer, New York, USA.

Chateauneuf, M. A. and Kreher, D. L. (2002).
On the state of strength-three covering arrays.
Journal of Combinatorial Design, 10(4):217–238.

Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. (1997).
The AETG system: An approach to testing based on combinatorial design.
IEEE Transactions on Software Engineering, 23:437–444.

Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C. (1996).
The combinatorial design approach to automatic test generation.
IEEE Software, 13(5):83–88.

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.

Referencias II

Kuhn, D. R., Lei, Y., and Kacker, R. N. (2008).
Practical combinatorial testing: Beyond pairwise.
IT Professional, 10(3):19–23.

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.

Lei, Y. and Tai, K. C. (1998).
In-parameter-order: A test generation strategy for pairwise testing.
In *Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering*, pages 254–261. IEEE Computer Society.

Myers, G. J. (1979).
The art of software testing.
John Wiley & Sons, first edition.

Myers, G. J. (2004).
The art of software testing.
Wiley, second edition.

Nurmela, K. J. (2004).
Upper bounds for covering arrays by tabu search.
Discrete Applied Mathematics, 138(1-2):143–152.

Rapps, S. and Weyuker, E. J. (1985).
Selecting software test data using data flow information.
IEEE Transactions on Software Engineering, 11(4):367–375.

Referencias III

Sloane, N. J. A. (1993).

Covering arrays and intersecting codes.
Journal of Combinatorial Designs, 1(1):51–63.

Stardom, J. (2001).

Metaheuristics and the search for covering and packing arrays.
Master's thesis, Simon Fraser University, Burnaby, Canada.

Torres-Jimenez, J. and Rodriguez-Tello, E. (2012).

New bounds for binary covering arrays using simulated annealing.
Information Sciences, 185(1):137–152.

Yan, J. and Zhang, J. (2008).

A backtracking search tool for constructing combinatorial test suites.
Journal of Systems and Software, 81(10):1681–1693.