Pruebas del Software

Índice

1. Introducción

- Proceso de prueba del software
  - Proceso mediante el cual se aplican una serie de métodos (ocasionalmente utilizando herramientas) que permiten obtener un conjunto de medidas para verificar y validar el funcionamiento requerido del software
- Verificar
  - Todo lo que hace, lo hace bien
  - Determinar si los productos de cada fase siguen las condiciones impuestas (seguimiento del procedimiento establecido)
- Validar
  - Hace todo lo que debería hacer
  - Determinar si un sistema o componente satisface los requisitos especificados

Principios básicos de las pruebas del software

- Una prueba exitosa es aquella que provoca que el sistema se desempeñe incorrectamente y así exponer un defecto
  - El objetivo principal de una prueba es descubrir algún error
  - El éxito de una prueba se mide en función de la capacidad de detectar un error que estaba oculto
- Un caso de prueba es bueno cuando su ejecución conlleva una probabilidad elevada de encontrar un error no detectado todavía
  - Caso de pruebas: especificación de una situación inicial del sistema, una serie de acciones a ejecutar (junto con los parámetros de entrada que sean necesarios) y un resultado esperado

¿Cómo se realizan las pruebas?

- La ejecución de las pruebas solo es la parte visible (40%), pero también es necesario planificarlas (20%) y prepararlas (40%)
- Formas de ejecutar las pruebas
  - De forma dinámica y explícita: se diseñan casos de prueba explícitos que implican la ejecución parcial o total del software con el fin de verificar y validar lo que hace
  - De forma dinámica e implícita: los casos de prueba anteriores revelan características del sistema no buscadas directamente en su diseño
    - ejemplos: evaluar ergonomía / facilidad de uso, rendimiento, …
  - De forma estática: se evalúa el software sin ejecutarlo

¿Quién ejecuta las pruebas? (I)

- Cualquiera de las partes involucradas en un sistema de información
  - Representantes del cliente: directivos, usuarios, administrador de sistemas, …
  - Representantes del equipo/empresa de desarrollo
  - Profesionales de las pruebas (dentro de empresa de desarrollo)
    - Probar es una disciplina profesional, requiere entrenamiento y gente preparada
      - No es una tarea a realizar por recién llegados
      - Es mejor saber como se hacen las cosas antes de saber como probarlas
      - Se persiguen errores, no personas
      - Cuando se encuentra un error hay que celebrarlo
      - Un buen probador es aquel que encuentra errores

¿Quién ejecuta las pruebas? (II)

- Se debe cultivar una actitud positiva sobre una creatividad destructiva
  - Se trata de una disciplina muy creativa
  - Los buenos probadores son metódicos exprimiendo los productos y llevándolos al límite de sus posibilidades

- Se recomienda que haya una mezcla de representantes y profesionales de las pruebas
  - Representantes del cliente: conocimiento de la temática
  - Representantes del desarrollo: conocimiento del sistema
  - Profesionales de las pruebas: conocimiento del proceso de pruebas e independencia en la medida que sea posible del proceso de desarrollo (se evita una simple verificación de que el software funciona correctamente, comprobando que ha sido concebido e interpretado correctamente)

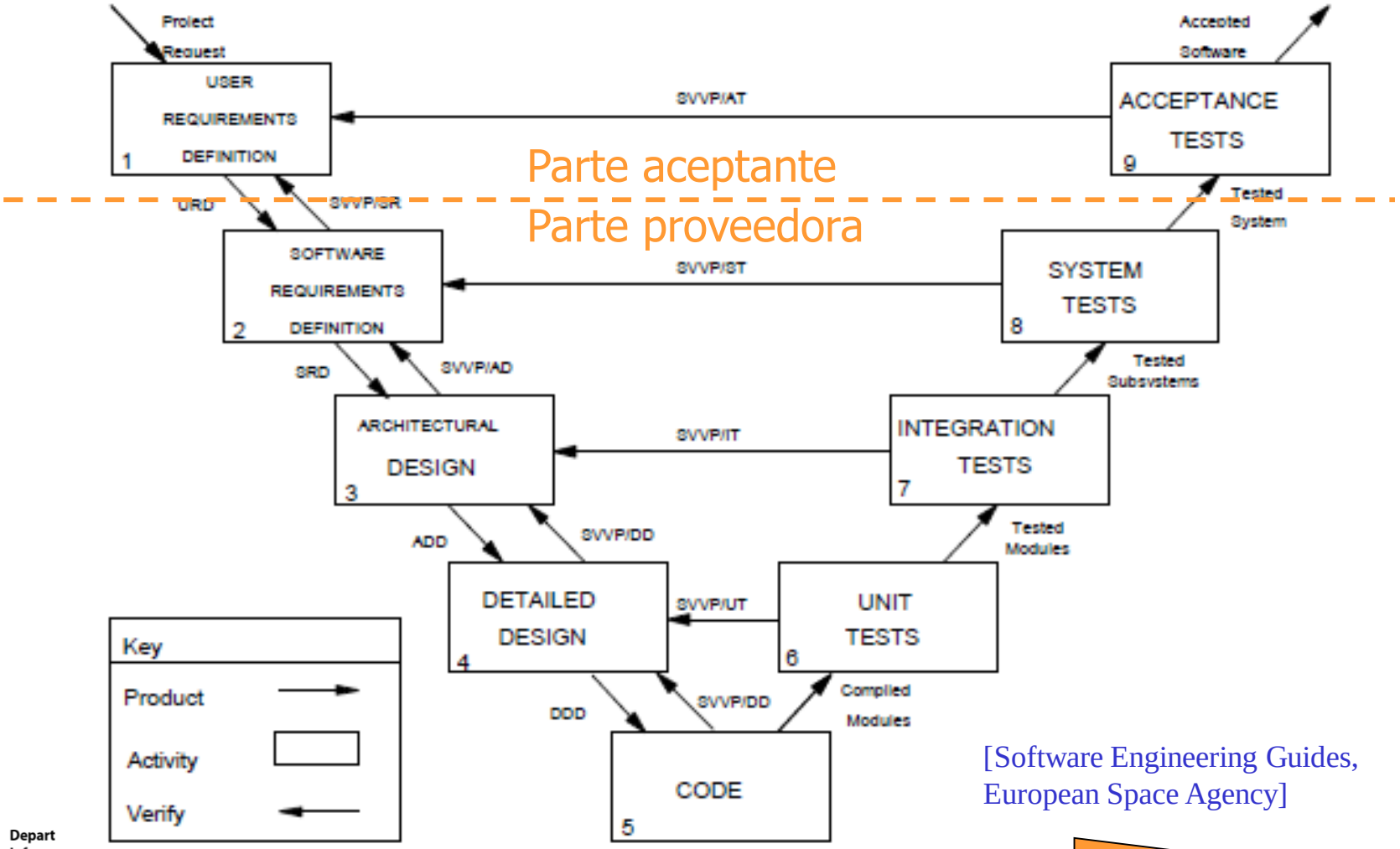
Niveles de pruebas y responsabilidades (I)

- En el contexto de las pruebas, se hace una distinción de responsabilidades
  - Parte proveedora (equipo/empresa de desarrollo)
    - Se asegura de que el producto entregado es el que se debería entregar
  - Parte aceptante (cliente, usuario, administrador, …)
    - Se asegura de que el producto recibido es el que se quería/necesitaba

Niveles de pruebas y responsabilidades (II)

- La distinción de responsabilidades se traduce en la agrupación de actividades de pruebas en niveles de pruebas que se gestionan y ejecutan colectivamente
  - Pruebas de desarrollo (parte proveedora)
    - Cumplimiento de especificaciones técnicas, …
  - Pruebas de sistema (parte proveedora)
    - Cumplimiento de requisitos funcionales, no funcionales, diseño técnico
  - Pruebas de aceptación (parte aceptante)
    - El producto cumple con las expectativas, …

Niveles de pruebas y responsabilidades (II)



[Software Engineering Guides, European Space Agency]

Preparación de las pruebas

Ejecución de las pruebas

2. Niveles de pruebas

- Pruebas de desarrollo
  - Pruebas unitarias
  - Pruebas de integración
  - Pruebas incrementales
- Pruebas de sistema
- Pruebas de aceptación
- Pruebas de regresión

2.1 Pruebas de desarrollo (I)

- Pruebas unitarias
  - Todos los componentes del sistema que se desarrollen se prueban individualmente para comprobar su correcto funcionamiento
  - Se realizan al crear cada módulo o componente individual de un sistema
- Pruebas de integración
  - Se prueba la integración entre los componentes del sistema para demostrar que se pueden encajar correctamente
  - Son realizadas sobre estos componentes agrupados, se examinan las interfaces para asegurar que estos componentes individuales son llamados cuando es necesario y que los datos que se transmiten entre dichos componentes son los requeridos

Pruebas de desarrollo (II)

- Problemas de realizar las pruebas unitarias de forma separada
  - Es necesario crear módulos auxiliares que simulen las acciones de los módulos invocados por el módulo que se está probando
  - Asimismo se han de crear módulos conductores para establecer las precondiciones necesarias, llamar al módulo objeto de la prueba y examinar los resultados de la prueba
- Posible solución: Prueba incremental
  - Combina pruebas unitarias y de integración
  - Consiste en agregar cada módulo o componente individual al conjunto de componentes existentes y el conjunto resultante se prueba
  - Se reduce el nº de módulos auxiliares y conductores
  - Se identifican mejor los errores (seguramente provenientes de un nuevo módulo o sus interfaces con otros módulos)

Pruebas de desarrollo (III)

- Estrategias para integrar componentes en pruebas incrementales
  - Estrategia de arriba a abajo (*top-down*)
    - El primer componente o módulo que se desarrolla y prueba es el que está arriba en la jerarquía, los módulos de nivel más bajo se sustituyen por módulos auxiliares para simular a los módulos que dichos módulos invocan
  - Estrategia de abajo a arriba (*bottom-up*)
    - En este caso se crean primero los componentes de más bajo nivel y se crean módulos conductores para simular a los módulos que los llaman
  - Estrategia combinada
    - Se desarrollarán partes del sistema de forma *top-down*, mientras que los componentes más críticos en el nivel más bajo se desarrollarán *bottom-up*

2.2. Pruebas del sistema y de aceptación (I)

- Pruebas globales
  - Una vez que se han probado los componentes individuales y se han integrado, se prueba el sistema globalmente para verificar que satisface todos los requisitos esperados
- Pruebas de sistema
  - Son pruebas globales realizadas por el equipo de desarrollo
- Pruebas de aceptación
  - Son también pruebas globales, pero determinadas por los usuarios/clientes, en lugar de serlo por el equipo de desarrollo, con el objeto de comprobar si el sistema es aceptable para ellos
  - Deben definirse de forma clara y explícita los resultados esperados: funciones del sistema que son críticas, tiempos de respuesta exigidos para el sistema, rendimiento del sistema en períodos críticos de alto volumen de transacciones
  - Un tipo especial de estas pruebas es el de la ejecución en paralelo con el viejo sistema, para comparar los resultados producidos por ambas ejecuciones

Tipos de pruebas globales (I)

- Pruebas funcionales
  - Consisten en determinar que se han cumplido los objetivos del sistema y que éste realiza correctamente todas las funciones que se han detallado en las especificaciones del sistema
- Pruebas de comunicaciones
  - Determinan que las interfaces entre los componentes del sistema funcionan adecuadamente
  - Esto se refiere tanto a las comunicaciones entre dispositivos remotos, como a las interfaces entre diferentes módulos dentro del mismo ordenador.
  - Asimismo se han de probar las interfaces persona/máquina

Tipos de pruebas globales (II)

- Pruebas de rendimiento
  - Consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema
- Pruebas de volumen
  - Consisten en examinar el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas
- Pruebas de sobrecarga (estrés)
  - Consisten en comprobar el funcionamiento del sistema en el umbral límite de los recursos, sometiéndole a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos
- Pruebas de disponibilidad de datos
  - Consisten en demostrar que el sistema puede recuperarse ante fallos, tanto de equipo físico como lógico, sin pérdidas indebidas de datos

Tipos de pruebas globales (III)

- Pruebas de facilidad de uso
  - Consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurarse de que se acomoda al modo habitual de trabajo de éstos, como para determinar si les facilita su trabajo a la hora de introducir datos en el sistema y de beneficiarse de las respuestas del mismo
- Pruebas de entorno
  - Consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno
- Pruebas de seguridad
  - Consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos

Tipos de pruebas globales (IV)

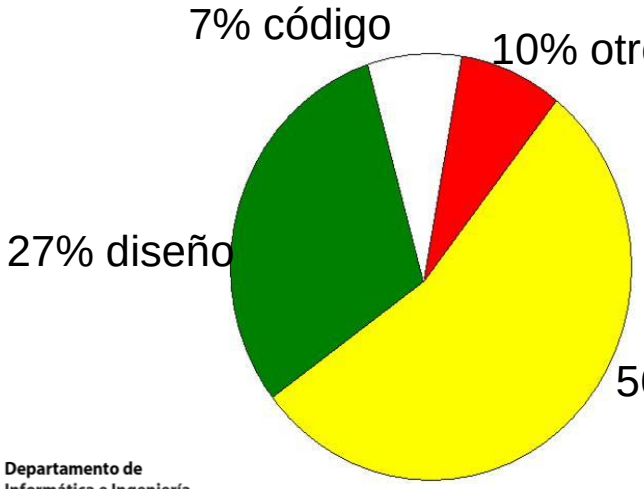
- Pruebas de comparación
  - En situaciones de elevado riesgo en las que la fiabilidad y corrección del software es un elemento crítico, se plantea el uso de software y hardware redundante
  - Software redundante
    → grupos separados desarrollan versiones independientes de una aplicación
  - Se prueba que las salidas coinciden
  - No es infalible:
    - Errores introducidos en la confección de las especificaciones
    - Coincidencia en un error

2.3. Pruebas de regresión

- Cada vez que se hace una modificación en el sistema tanto para corregir un error como para añadir una mejora, se han de re-ejecutar una parte de las pruebas previamente realizadas
  - "La probabilidad de encontrar errores adicionales en una sección del software es proporcional al número de errores ya encontrados en la misma sección"
- No basta con probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan; cualquier componente del sistema puede haber sido afectado por el cambio y se ha de tener cuidado de no introducir nuevos errores
- Aunque la mayoría de las pruebas de regresión tienen lugar en la fase de mantenimiento, algunas pueden tener lugar en la fase de implantación del sistema
- Por último hay que tener en cuenta que, a medida que se van resolviendo problemas detectados durante las pruebas, puede ocurrir que haya que rehacer las pruebas realizadas al principio

3. Pruebas estáticas (I)

- El proceso de prueba del software tiene su propio ciclo de vida
  - Comienza en la fase de captura de requisitos y evoluciona en paralelo con el proceso de desarrollo
- El coste de los errores se minimiza si se detectan en la misma fase en la que se introducen y se evita su transferencia a otras fases posteriores
  - Más de la mitad de los errores son introducidos el la captura de requisitos y análisis



Distribución de los defectos
[J. Martin, 1984. *An Information System Manifesto,* Prentice Hall]
[R. Bender, 2009. *Requirements Based Testing Process Overview,* Benders RBT Inc.]

Pruebas estáticas (II)

- Cada fase de desarrollo puede llevar asociada una actividad de prueba
- El primer paso en el proceso de prueba del software puede consistir en inspeccionar las documentaciones asociadas a cada fase con el fin de comprobar que se ajustan a lo deseable
- Pruebas estáticas: se evalúa el software sin ejecutarlo
- Para cada tipo de documento (Requisitos, análisis, código, documentación general) se puede establecer una lista de elementos a comprobar
  - Por ejemplo, ya vimos una lista para la validación de requisitos en la fase de requisitos

Inspección de código fuente (I)

- Aparte de seguir guías de estilo de codificación, es conveniente inspeccionar el código para verificar la ausencia de algunos problemas (no detectados normalmente por un compilador):

Errores en la referencia a datos	Variables no referenciadas o no inicializadas
Errores en la declaración de datos	Variables con el mismo nombre
Errores de computación	Desbordamientos
Errores en comparaciones	Comparación entre variables de tipos inconsistentes
Errores en el control del flujo	Posibles salidas tempranas de bucles
Errores en la interfaz	Parámetros no utilizados, efectos laterales, accesos a variables globales
Errores de entrada/salida	Errores ortográficos o gramaticales en las salidas textuales

Ejemplo de inspección de código fuente(I)

```
BOOL CALLBACK newTicketManager (
   HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
 int idCode;
 int status;
 Ticket* currentTicket;
 char auxBuff[150];
 switch (message) {
  case WM_COMMAND:
      if(LOWORD(wParam) == IDTOTAL){
       sprintf(auxBuff,"El total del ticket es: %d\n¿Finalizar la venta?", ticketValue(currentTicket));
       /* ticketValue calcula el total del ticket que le paso */
       idCode = MessageBox(hDlg,auxBuff,"Cuestion a plantear", MB_YESNO|MB_ICONQUESTION);
       if(idCode == IDNO) {
              return(TRUE); break;
       }
       /* Guarda el ticket en la base de datos */
       addElementToTicketDataBase(*currentTicket);
       deleteTicket(currentTicket);
       currentTicket = newTicket();
```

Variable que no se utiliza

Variable no inicializada
Error en la referencia a datos

Error de entrada/salida

Ejemplo de inspección de código fuente (II)

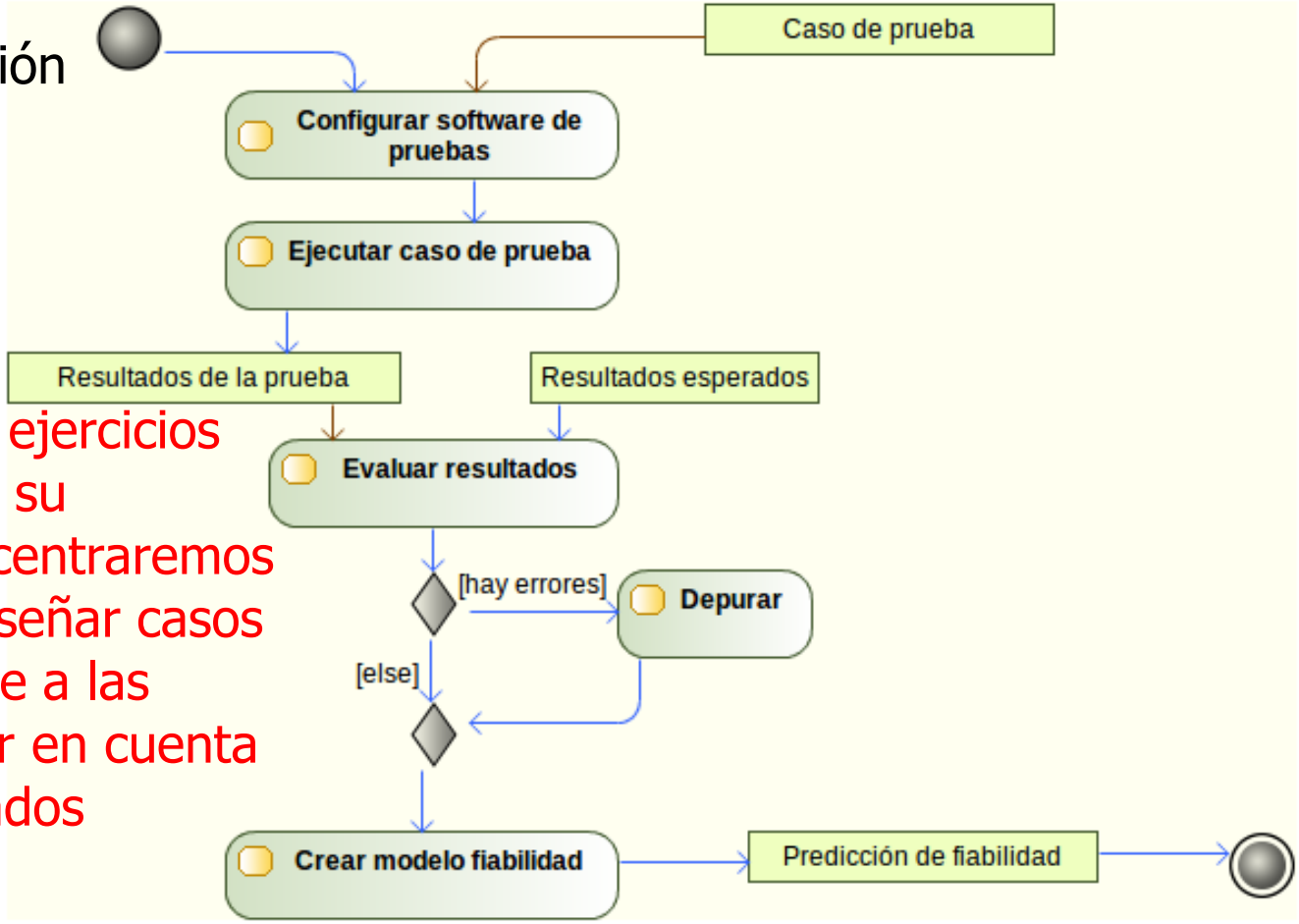
```
                /* Limpia las ventanas de edición */
                SetDlgItemText(hDlg, IDC_PRODUCT_PRICE, "");
                SetDlgItemText(hDlg, IDC_PRODUCT_UNITS, "");
                SetDlgItemText(hDlg, IDC_PRODUCT_PRICE, "");
                SetDlgItemText(hDlg, IDC_PRODUCT_DESCRIPTION, "");
                SetDlgItemText(hDlg, IDC_ITEM_TICKET_VALUE, "");
                SetDlgItemText(hDlg, IDC_TICKET_VALUE, "");
                SendMessage (GetDlgItem (hDlg, IDC_TICKET_LIST),LB_RESETCONTENT,lParam,0);
                EnableWindow(GetDlgItem(hDlg, IDTOTAL), FALSE);
                EnableWindow(GetDlgItem(hDlg, IDMINUS), FALSE);
                strcpy(auxBuff,"Fecha: ");
                strcat(auxBuff, timeTypeToString(currentTicket->date));
                SetDlgItemText(hDlg, IDC_TICKET_DATE, auxBuff);
                return(TRUE);
        }else return(FALSE);
        break;
    case WM_DESTROY:
        if(currentTicket = 0)
          MessageBox(hDlg,"El ticket actual ha sido abandonado",
             "¡¡¡ Información !!!", MB_OK|MB_ICONEXCLAMATION);
          return(FALSE); break;
     default:
         return(FALSE); break;
   }
   return(TRUE);
 }
```

Error de entrada/salida

Error de comparación

4. Pruebas dinámicas

- Se diseñan casos de prueba explícitos que implican la ejecución parcial o total del software con el fin de verificar y validar lo que hace
- Flujo de información de una prueba



- Nota: en algunos ejercicios de pruebas, dada su abstracción, nos centraremos únicamente en diseñar casos de prueba en base a las entradas sin tener en cuenta resultados esperados

Técnicas para diseñar casos de pruebas

- Pruebas de caja blanca (basadas en la estructura)
  - Requieren el esfuerzo de examinar la estructura interna del software
  - Se selecciona un conjunto de caminos lógicos y se generan casos de prueba, determinando los valores específicos que definen la ejecución de esos caminos seleccionados
- Pruebas de caja negra (basadas en las especificaciones)
  - Se diseñan los casos de prueba y los datos de prueba a partir de las especificaciones funcionales (entradas y salidas del sistema)
  - Se busca probar: las funciones realizadas por el sistema, el cumplimiento de los objetivos del sistema, las reacciones del sistema ante los estímulos exteriores
- No son excluyentes sino complementarias

Finalización de las pruebas

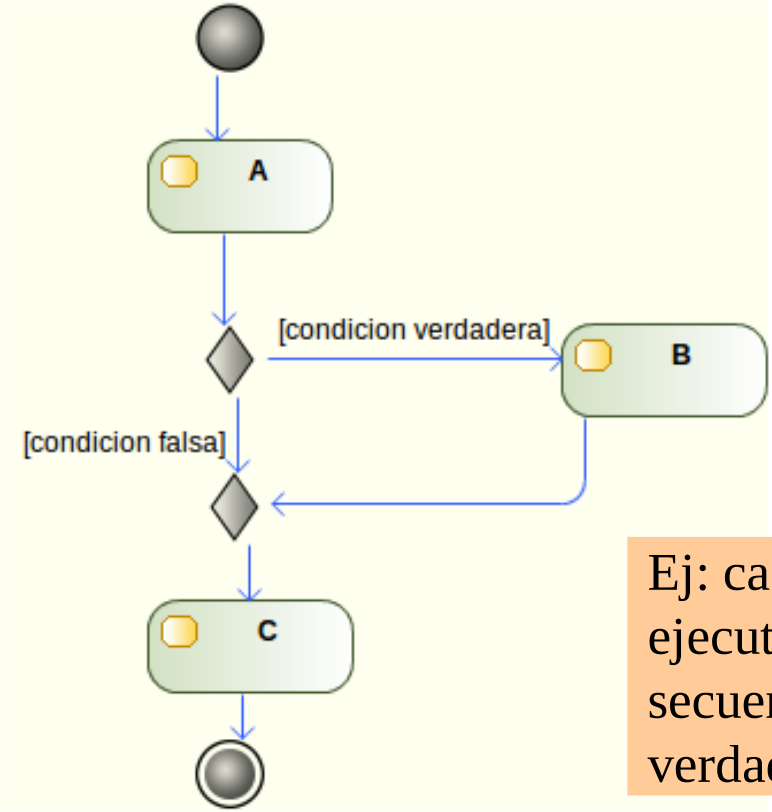
- *"Testing can only show the presence of bugs, not their absence"* [Dijkstra]
- Es difícil asegurar que el último error detectado, era el único que quedaba
- Métodos habituales para finalizar con las pruebas:
  - Terminar el proceso de pruebas cuando hemos consumido el tiempo planificado
  - Terminar las pruebas cuando todos los casos de prueba se ejecutan sin detectar errores
  - Realizar estimaciones sobre el nº de errores del software y finalizar cuando se haya alcanzado este nº
    - Es necesario contar con una historia o experiencia previa que permita definirlas (modelos de fiabilidad)

4.1. Pruebas de caja blanca

- Algunas técnicas:
  - Prueba de cobertura de sentencias
  - Prueba de cobertura de condición
  - Prueba de cobertura de decisión
  - Prueba de cobertura de condición múltiple
  - Prueba de caminos
  - Prueba de bucles

Técnica: Prueba de cobertura de sentencias (I)

- Consiste en generar casos de prueba que permiten probar TODAS Y CADA UNA de las sentencias dentro de un módulo, al menos una vez



Ej: caso de prueba que permita ejecutar camino conteniendo secuencia A,B,C (condición verdadera)

Prueba de cobertura de sentencias (II)

- Esta cobertura requiere que se ejecute por lo menos una vez cada sentencia del programa.

```
begin
    get(x);
    if x < 0 then
        x := -x;
    end if;
    put(x, 1);
end;
```

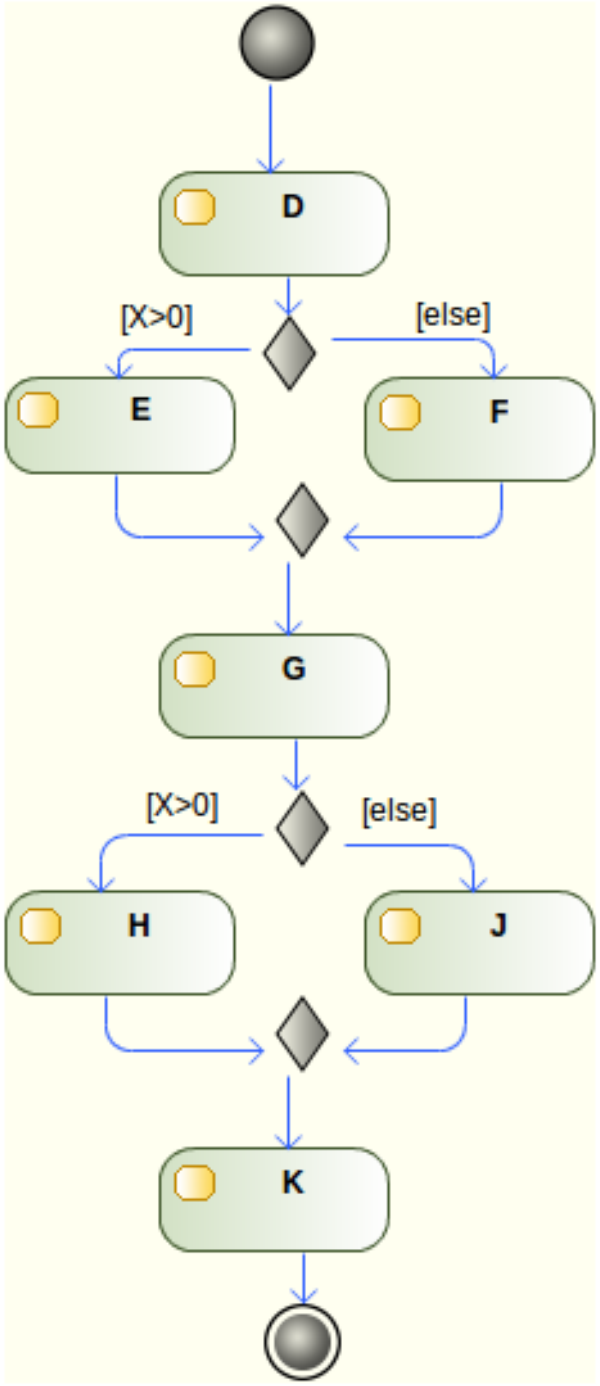
- Es un criterio débil
  - No se comprueban ambas vertientes de las condiciones
    - Ejemplo: en el programa anterior x=-1 sería el único caso de prueba necesario para cubrir todas las sentencias

Técnica: Prueba de cobertura de condición (I)

- Una decisión es una combinación de una o más condiciones
- La prueba de cobertura de condición consiste en escribir casos de prueba de modo que cada condición en una decisión tenga, al menos una vez, todos los resultados posibles.
- Este tipo de pruebas es más fuerte que el anterior (cobertura de sentencias).
- Los tipos de errores que pueden aparecer en una condición son los siguientes:
  - Existe un error en un operador lógico (que sobra, falta o no es el que corresponde)
  - Existe un error en un paréntesis lógico (cambiando el significado de los operadores involucrados)
  - Existe un error en un operador relacional (operadores de comparación de igualdad, menor o igual, etc.)
  - Existe un error en una expresión aritmética

Prueba de cobertura de condición (II)

- Hay que tener cuidado con los casos de prueba elegidos (y sus caminos asociados)
  - Aunque a priori garanticen los valores posibles de cada condición, puede que no se puedan ejecutar
  - Ejemplo:
    - ¿[D,E,G,H,K] y [D,F,G,J,K] ?
    - o ¿[D,E,G,J,K] y [D,F,G,H,K]?
    - Si el algoritmo no altera el valor de la variable X, la segunda alternativa parece poco probable



Pruebas de cobertura de condición (III)

- Hay que ser cuidadoso con la elección de los casos de prueba porque aunque se garantice la distinta evaluación de las condiciones, puede ocurrir que alguna cláusula de la decisión no sea ejecutada
- Ejemplo:

```
if ((numeroLibros>8) or (precioTotal>250€)) then
 precioFinal = aplicarDescuento(precioTotal);
else
 precioFinal = precioTotal;
end if;
```

  - Caso de pruebas propuestos (cumplen con criterio de cobertura de condición):

ID	Entradas		Condiciones		Resultado esperado
	numeroLibros	precioTotal	numeroLibros>8	precioTotal>250	aplicarDescuento
1	9	50 €	V	F	V
2	5	300 €	F	V	V

Técnica: Prueba de cobertura de decisión (o de ramificación)

- Consiste en escribir casos de prueba suficientes para que cada decisión, por lo menos una vez, tenga un resultado verdadero y uno falso

ID	Entradas		Condiciones		Resultado esperado
	numeroLibros	**precioTotal**	**numeroLibros>8**	**precioTotal>250**	**aplicarDescuento**
1	5	300 €	F	V	V
2	5	50 €	F	F	F

- Posible debilidad:
  - En sentencias condicionales compuestas puede quedar enmascarada una de las condiciones

Técnica: Prueba de cobertura de condición múltiple

- Consiste en escribir casos de prueba que tengan en cuenta todas las posibles combinaciones de los resultados de todas las condiciones en cada decisión
- Satisface los criterios de las pruebas de condición y decisión

ID	Entradas		Condiciones		Resultado esperado
	numeroLibros	precioTotal	numeroLibros>8	precioTotal>250	aplicarDescuento
1	9	300 €	V	V	V
2	9	50 €	V	F	V
3	5	300 €	F	V	V
4	5	50 €	F	F	F

- Posible problema: gran número de casos de prueba

Técnica: Prueba de caminos (prueba del camino básico)

- Esta técnica se basa en obtener una medida de la complejidad lógica de un programa y usarla para definir un conjunto básico de caminos que garanticen que cada arista de procesamiento del programa se ejecuta al menos una vez
- Complejidad ciclomática
  - Medida del software que aporta una medida cuantitativa de la complejidad de un programa
  - Determina el número de caminos independientes del conjunto básico del programa y da una cota superior para el número de pruebas que se deben realizar para asegurar que al menos cada arista de procesamiento se va a ejecutar una vez
  - Camino independiente es aquel que introduce por lo menos una arista que no estaba considerada en el conjunto de caminos independientes calculados hasta ese momento

Procedimiento

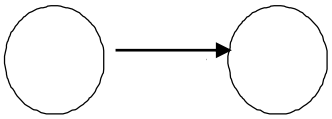
- Derivación de casos de prueba en función de la complejidad ciclomática
- Pasos
  1. A partir del diseño (diagrama de actividad) o del código fuente se dibuja el grafo de flujo asociado
  2. Se calcula la complejidad ciclomática del grafo
  3. Se determina el conjunto básico de caminos
  4. Se preparan casos de prueba que obliguen a la ejecución de cada camino del conjunto

Grafos de flujo

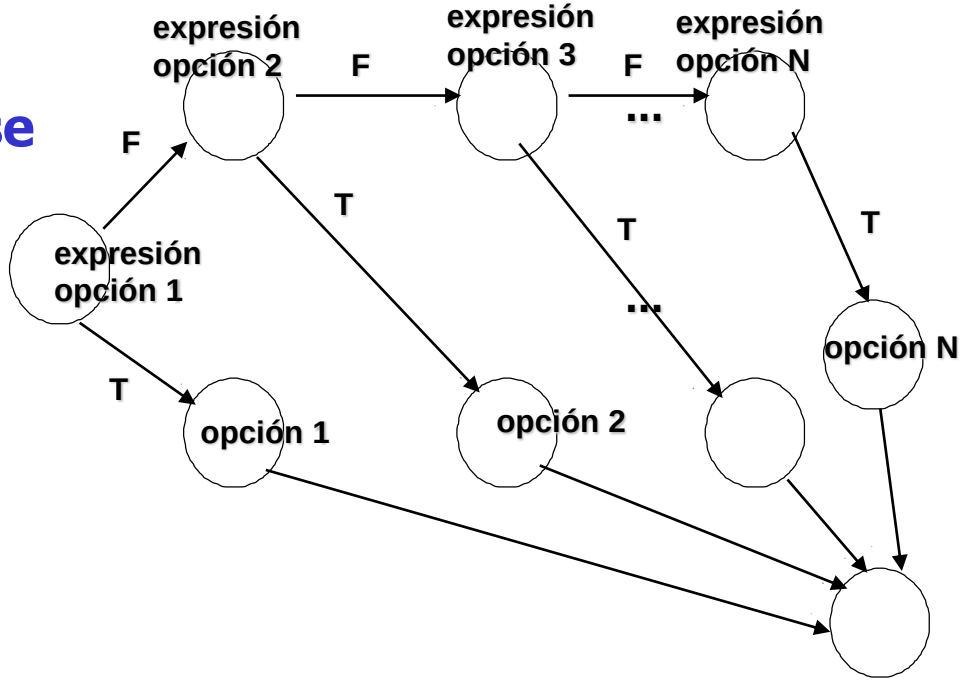
- Representan el flujo de control del programa
- Está compuesto por:
  - Nodos
    - bloques de código (una o más sentencias)
  - Arcos/Aristas
    - representan el flujo de control
    - son las conexiones entre los bloques de código
  - Regiones
    - Áreas delimitadas por aristas y nodos
- Nodos predicado
  - Nodo que contiene una condición
  - Se caracteriza porque de él salen dos o más aristas
- Una de las características para comprobar si un grafo de flujo está bien diseñado es que los únicos nodos de los que pueden partir dos aristas son los nodos predicados

Paso 1: Construir grafos de flujo (transformación de código)

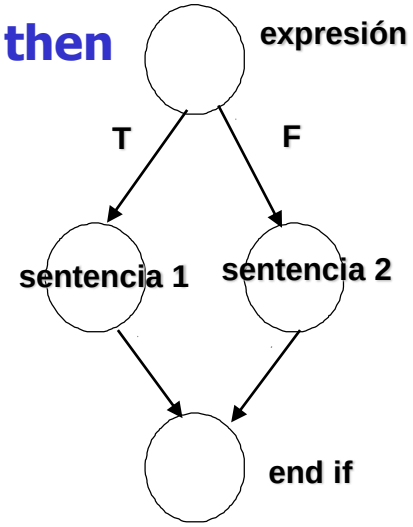
secuencia



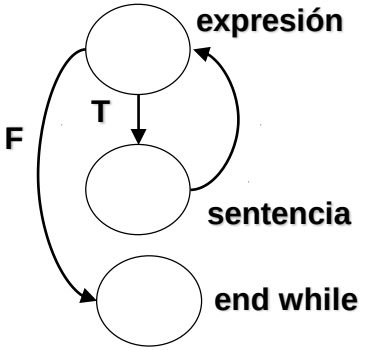
case



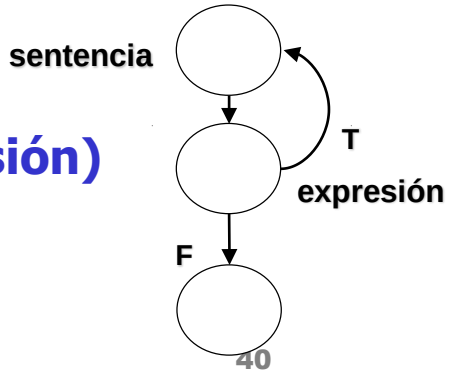
if (expresión) then
sentencia 1
else
sentencia 2
end if



while (expresión)
sentencia
end while

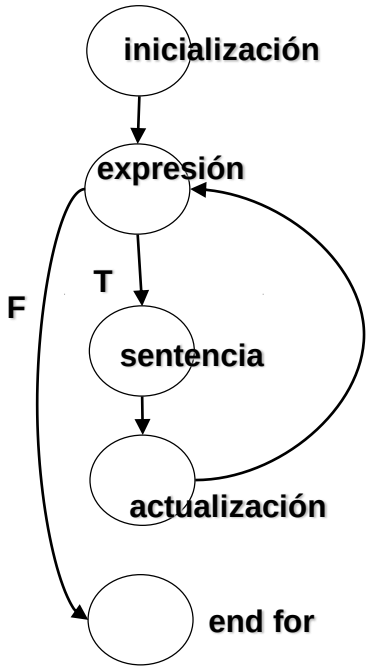


do
sentencia
while (expresión)

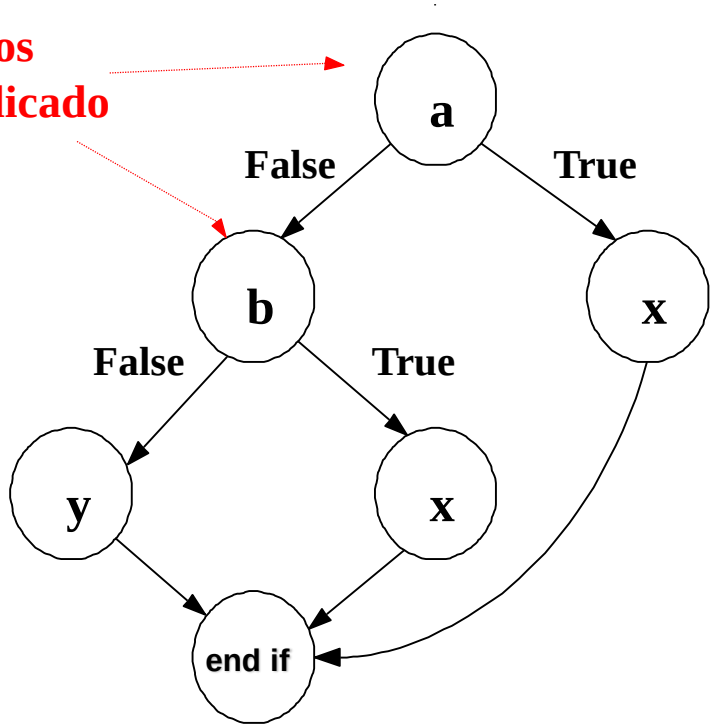


Paso 1: Construir grafos de flujo (transformación de código)

for (inicialización; expresión; actualización)
sentencia
end for



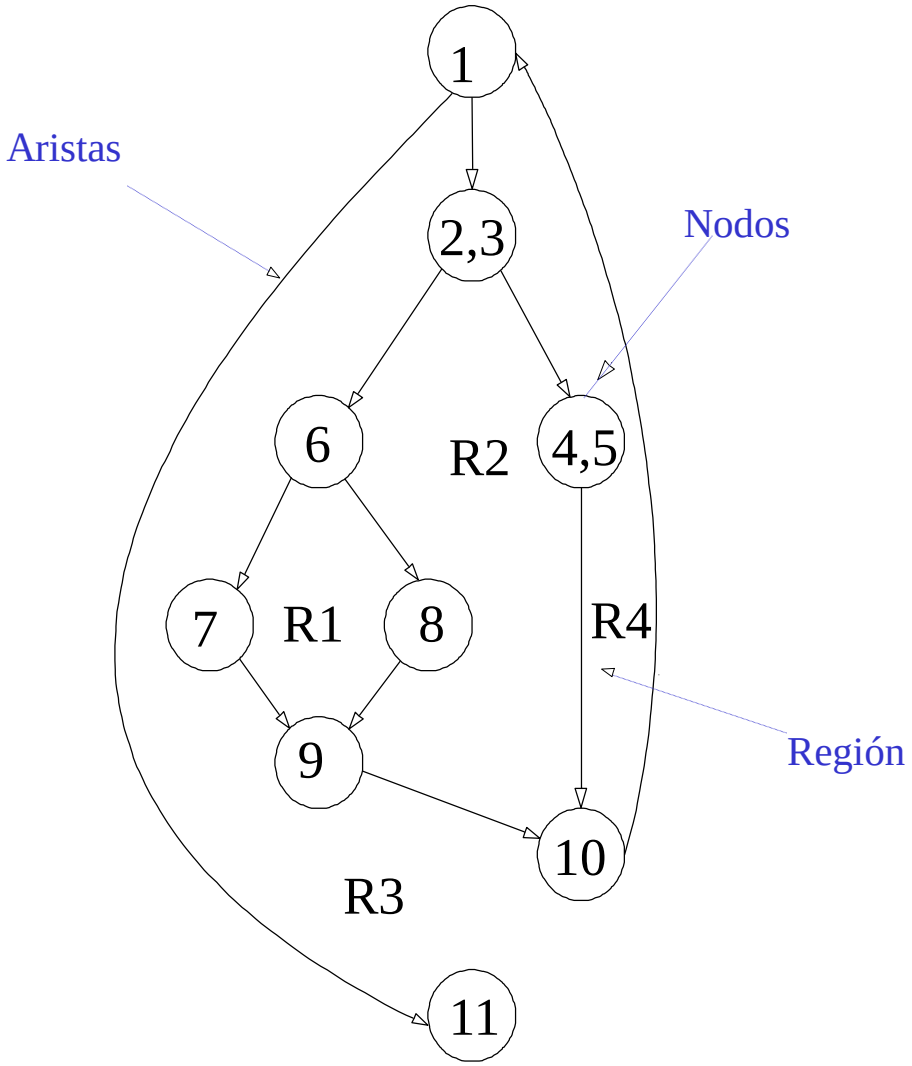
if *a* or *b* then
x
else
y
end if



Ejemplo de construcción de grafos de flujo (I)

```
begin                                   1
   while not end_of_file(f) loop
    2 read(f, n);  3
      if n mod 2 = 0 then
       4  put("Leído un número par: ");
       5  put(n, 1);
      else                    6
         if esPrimo(n) then
          7     put_line("Leído un número primo");
         else
          8     put_line("Leído un impar compuesto");
         end if;  9
      end if;  10
   end loop;
end;  11
```

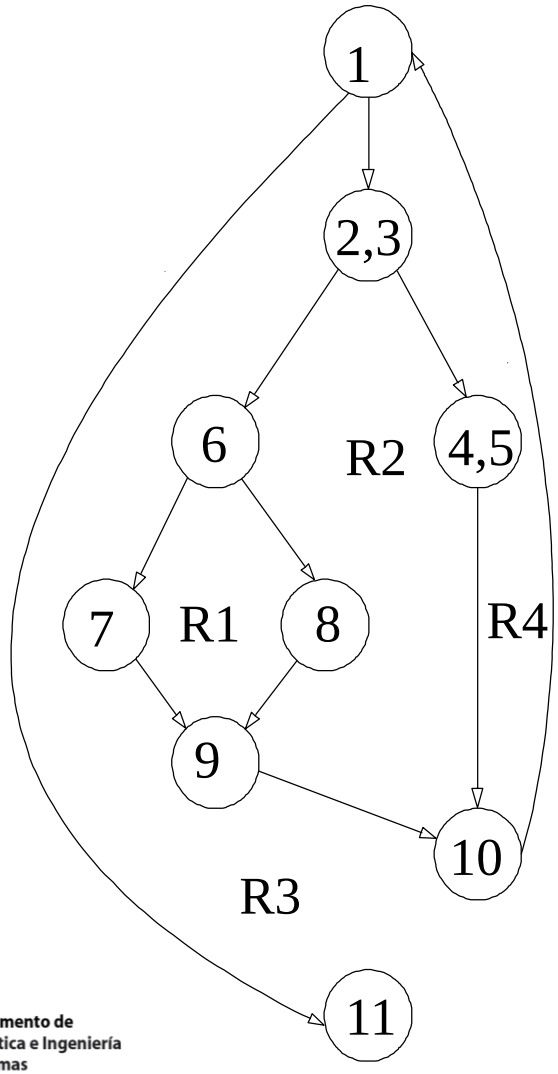
Ejemplo de construcción de grafos de flujo (II)



Paso 2: Cálculo de la complejidad ciclomática

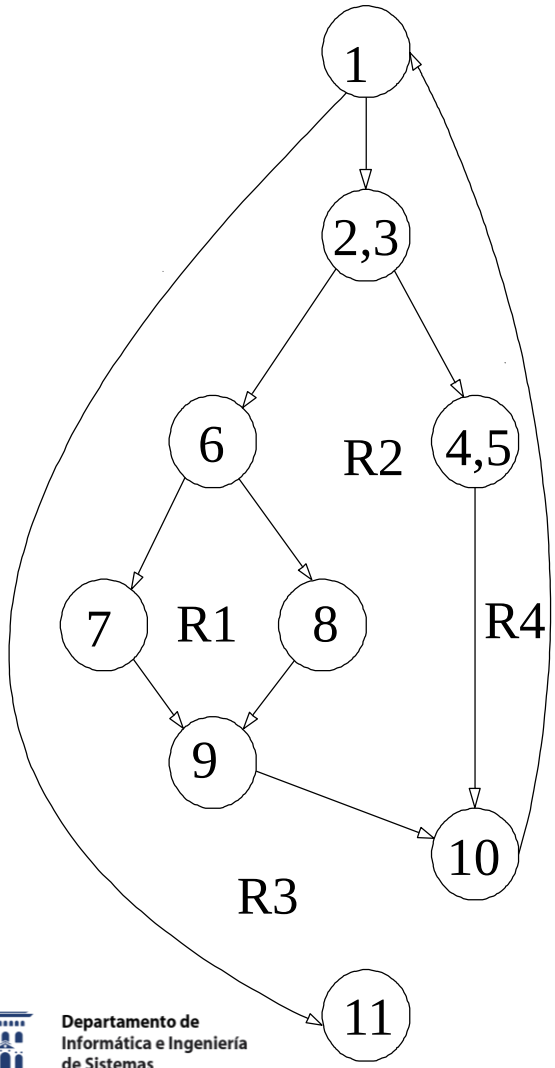
- La complejidad ciclomática de un grafo de flujo V(G) establece el número de caminos independientes
- Puede calcularse de tres formas alternativas:
  - El número de regiones del grafo de flujo
  - V(G) = A - N + 2
    - donde A es el número de aristas
    - y N es el número de nodos
  - V(G) = P + 1
    - donde P es el número de nodos predicado

Ejemplo de cálculo de la complejidad ciclomática



- V(G) = 4
  - El grafo de la figura tiene cuatro regiones.
  - 11 aristas - 9 nodos + 2 = 4
  - 3 nodos predicado + 1 = 4

Determinar el conjunto de caminos básicos (paso 3) y preparar los casos de prueba (paso 4)

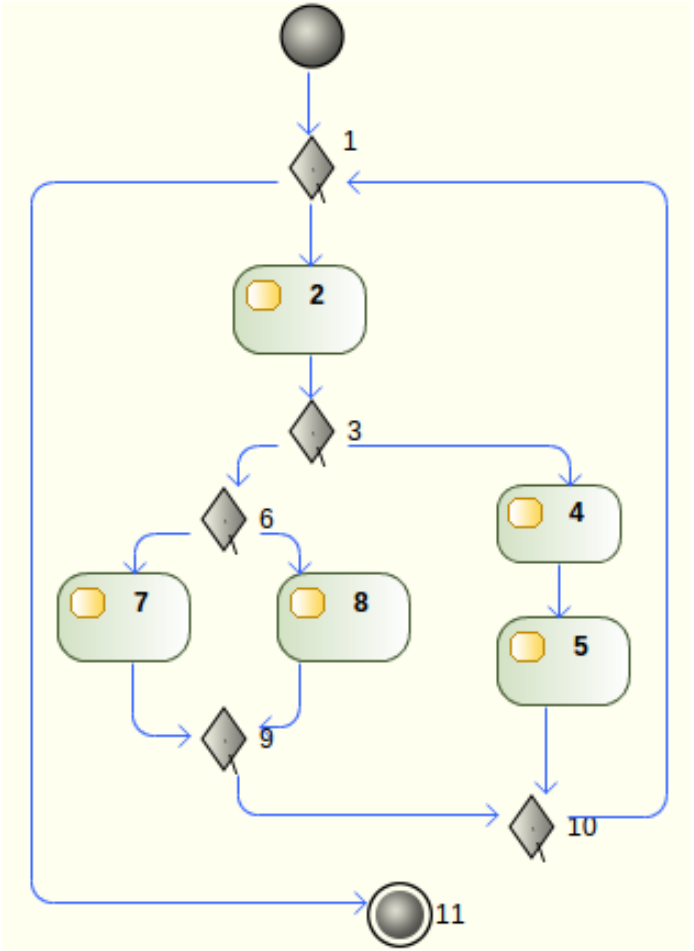


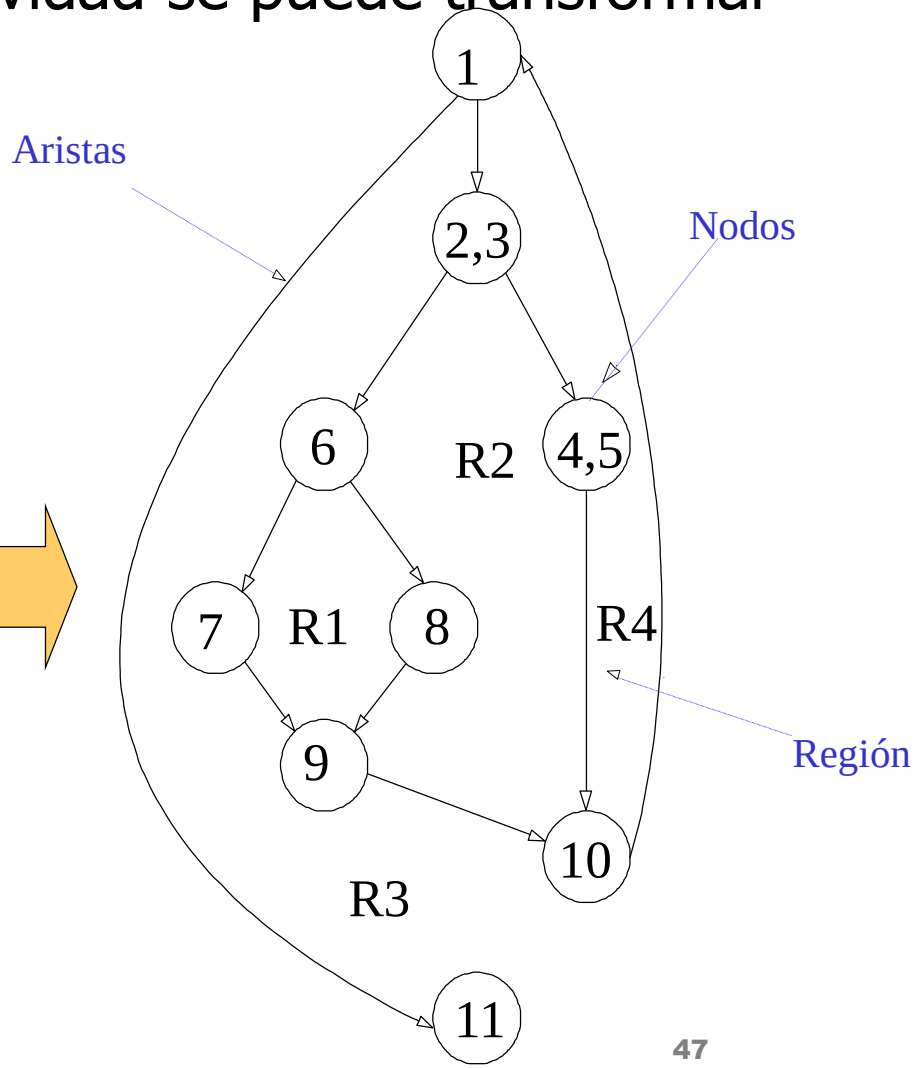
- Un conjunto de caminos independientes
  - Camino 1: 1-11
  - Camino 2: 1-2,3-4,5-10-1-11
  - Camino 3: 1-2,3-6-8-9-10-1-11
  - Camino 4: 1-2,3-6-7-9-10-1-11
- El camino
  1-2,3-4,5-10-1-2,3-6-8-9-10-1-11
  no se considera un camino independiente
  - Es simplemente una combinación de caminos ya especificados
- Los cuatro caminos anteriores constituyen un conjunto básico para el grafo
- Casos de prueba para cada camino:

Camino	Entrada (f)	Resultado esperado (pantalla)
1	{} (fichero vacío)	<>
2	{2}	Leído un número par: 2
3	{9}	Leído un impar compuesto
4	{3}	Leído un número primo

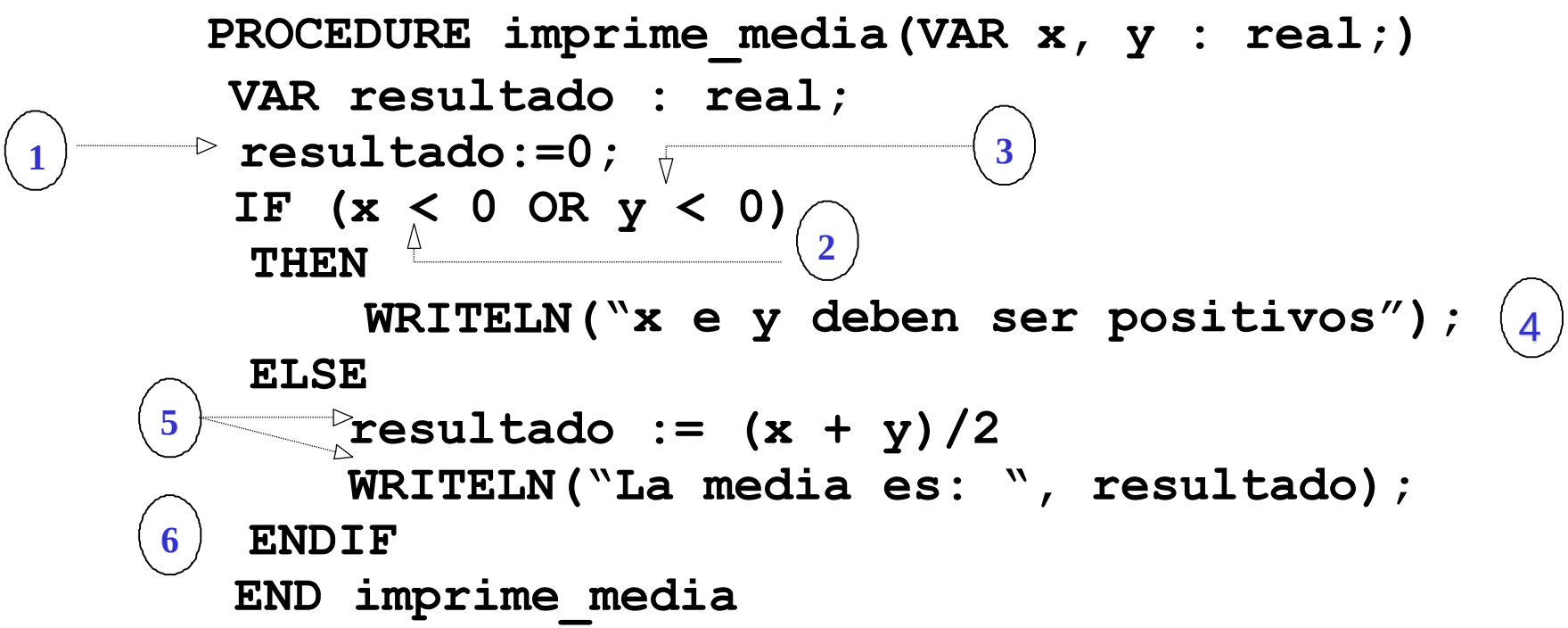
Técnica aplicable a cualquier flujo de trabajo (no solo código o descripción de operaciones)

- Cualquier diagrama de actividad se puede transformar a un grafo de flujo



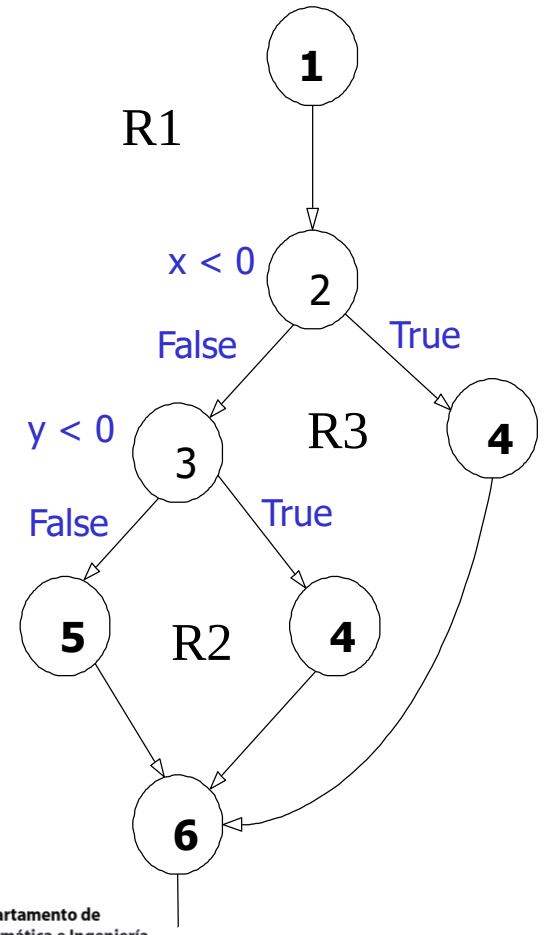


Un ejemplo completo (I)



Un ejemplo completo (II)

- 1. Construir grafo de flujo

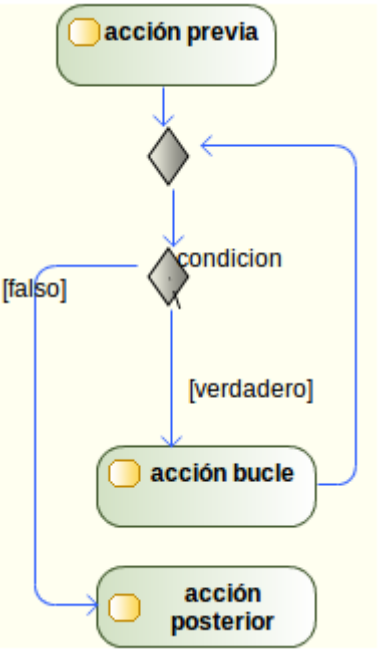


- 2. Calcular complejidad
  - V(G) = 2+1 = 3
- 3. Definir caminos básicos:
  - Camino 1: 1-2-3-5-6
  - Camino 2: 1-2-4-6
  - Camino 3: 1-2-3-4-6
- 4. Preparar casos de prueba:

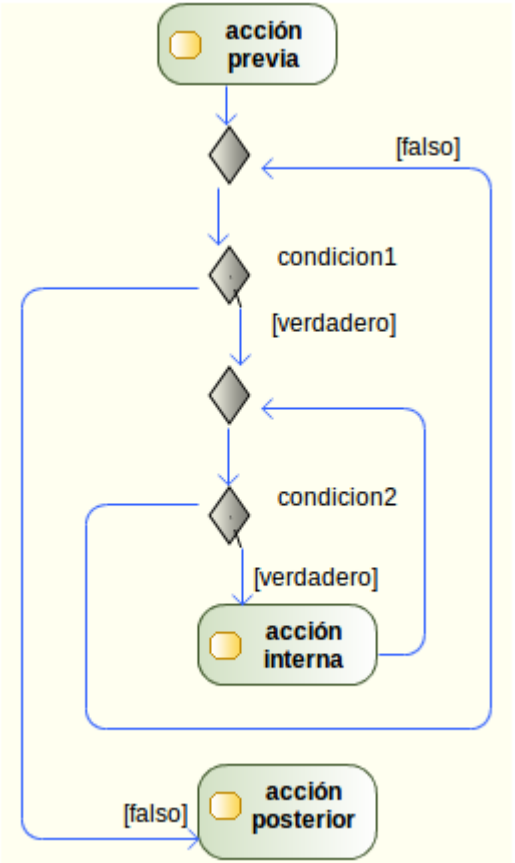
Camino	**Entradas**		**Resultado esperado (pantalla)**
	x	**y**	
1 [Escoger algún *x* e *y* tales que se cumpla *x>=0* AND *y>=0*]	3	3	La media es: 3
2 [Escoger algún x tal que se cumpla x<0]	-3	3	x e y deben ser positivos
3 [Escoger algún x e y tales que se cumpla x>=0 AND y<0]	3	-3	x e y deben ser positivos

Técnica: Prueba de bucles

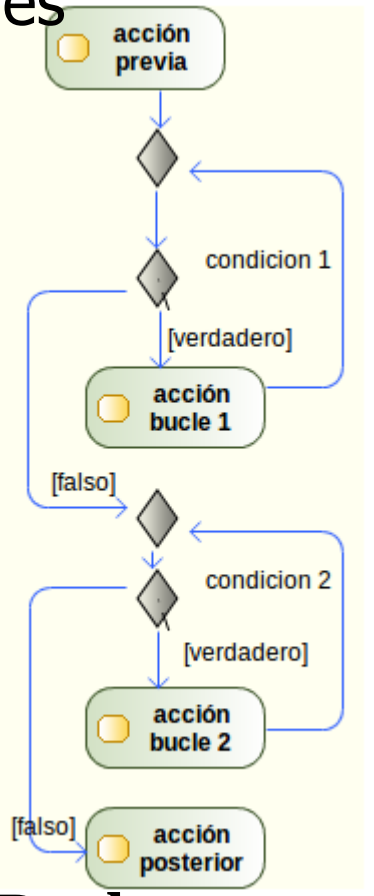
- Técnica que se centra únicamente en la corrección de las construcciones de bucles



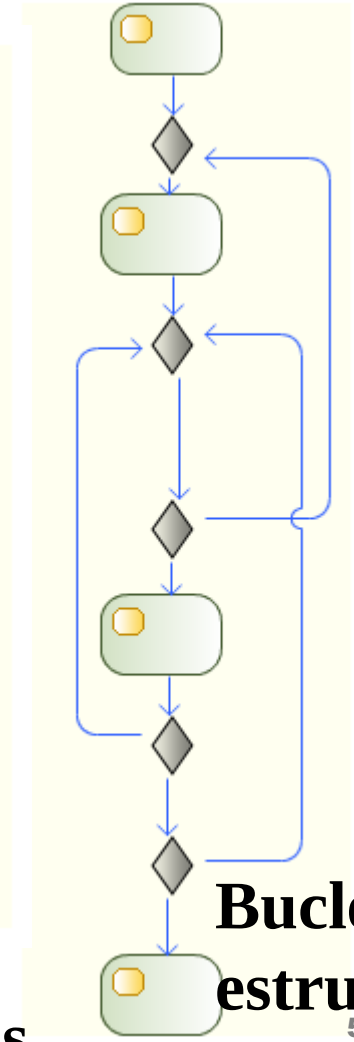
Bucles Simples



Bucles anidados



Bucles concatenados



Bucles no estructurados

Procedimiento

- Bucles simples. Se deben aplicar las siguientes pruebas:
  - Pasar por alto el bucle
  - Pasar una sola vez el bucle
  - Pasar dos veces el bucle
  - Hacer un número medio *m* iteraciones (con $m < n$)
  - Hacer $n - 1$ y *n* iteraciones por el bucle (donde *n* es máximo número posible de iteraciones para el bucle)
- Bucles anidados
  - Realizar las pruebas de bucle simple al bucle más interior estableciendo los demás bucles en sus valores mínimos
  - Avanzar hacia fuera confeccionando pruebas para el siguiente bucle manteniendo todos los externos con valores mínimos y los internos con valores medios
  - Continuar hasta terminar de probar bucles

Ejercicio de bucles anidados (I)

```
int calcula(int primero, int segundo, int tercero) {
      int j = primero;
      int k;
      int result = 0;
      while (j < 100) {
            for (int i=tercero; i>0; i--) {
                  k = segundo;
                  while (k < 150) {
                        result++;
                        k++;
                  }
                  result++;
            }
            result++;
            j++;
      }
      return result;
}
```

Ejercicio de bucles anidados (II)

- Plantilla para casos de prueba

Bucle a probar	Objetivos:			Entradas			Resultado esperado
	en bucle externo	en bucle intermedio	en bucle interno	parametro 1	parametro 2	parametro 3	
Bucle interno	Nº mínimo de iteraciones	Nº mínimo de iteraciones	saltar				
			pasar 1 vez				
			pasar 2 veces				
			pasar m veces				
			pasar n-1 veces				
			pasar n veces				
Bucle intermedio	Nº mínimo de iteraciones	saltar	Nº medio de iteraciones				
		pasar 1 vez					
		pasar 2 veces					
		pasar m veces					
		pasar n-1 veces					
		pasar n veces					
Bucle externo	saltar	Nº medio de iteraciones	Nº medio de iteraciones				
	pasar 1 vez						
	pasar 2 veces						
	pasar m veces						
	pasar n-1 veces						
	pasar n veces						

Ejercicio de bucles anidados (III)

Solución

Bucle a probar	Objetivos: en bucle externo	en bucle intermedio	en bucle interno	Entradas primero=	segundo=	tercero=	Resultado esperado
Bucle interno	N.º mínimo de iteraciones	N.º mínimo de iteraciones	saltar	99	150 (o mayor)	1	2
			pasar 1 vez	99	149	1	3
			pasar 2 veces	99	148	1	4
			pasar m veces	99	75	1	77
			pasar n-1 veces	no hay nº máximo iteraciones			
			pasar n veces	no hay nº máximo iteraciones			
Bucle intermedio	N.º mínimo de iteraciones	saltar	N.º medio de iteraciones	99	75	0 (o menor)	1
		pasar 1 vez		99	75	1	77
		pasar 2 veces		99	75	2	153
		pasar m veces		99	75	100	7601
		pasar n-1 veces		no hay nº máximo iteraciones			
		pasar n veces		no hay nº máximo iteraciones			
Bucle externo	saltar	N.º medio de iteraciones	N.º medio de iteraciones	100 (o mayor)	75	100	0
	pasar 1 vez			99	75	100	7601
	pasar 2 veces			98	75	100	15202
	pasar m veces			50	75	100	380050
	pasar n-1 veces			no hay nº máximo iteraciones			
	pasar n veces			no hay nº máximo iteraciones			

Bucles concatenados

- Procedimiento
  - Se pueden probar igual que los bucles simples si son independientes entre sí
  - y siguiendo la técnica de los bucles anidados si existe cualquier relación entre los diferentes bucles concatenados
    - Por ejemplo, si hay dos bucles concatenados y se usa el contador del bucle 1 como valor inicial del bucle 2, entonces los bucles no son independientes → se aplicará el enfoque para bucles anidados

Ejercicio de bucles concatenados (I)

- Tratamiento como bucles independientes

```
int calcula(int f) {
  int k;
  int result = 0;
  if (f >= 0) {
    k = f;
    do {
      result++;
      k++;
    } while (k>=33 && k<150)
  } else {
    k = f;
    do {
      result++;
      k++;
    } while (k>=-666 && k<0)
  }
  return result;
}
```

Ejercicio de bucles concatenados (II)

- Bucle superior, f ≥ 0

Objetivos	Entrada f	Resultado esperado
saltar	no se puede saltar	
pasar 1 vez	149 (f>=149, 0<=f<=31)	1
pasar 2 veces	148	2
pasar m veces	100	50
pasar n-1 veces	33	117
pasar n veces	32	118

- Bucle inferior, f<0

Objetivos	Entrada f	Resultado esperado
saltar	no se puede saltar	
pasar 1 vez	-1 (f<=-668)	1
pasar 2 veces	-2	2
pasar m veces	-300	300
pasar n-1 veces	-666	666
pasar n veces	-667	667

Ejercicio de bucles concatenados (IV)

- Tratamiento como bucles anidados

```
int calcula(int primero, int segundo) {
  int i,j;
  int sum = 0;

  i = primero;
  while (i<50) {
    i++;
    sum++;
  }

  j=i;
  while (j<segundo) {
    j++;
    sum++;
  }
  return sum;
}
```

Bucle a probar	Objetivos		Entradas		Resultado esperado
	en bucle superior	en bucle inferior	primero	segundo	
Bucle inferior	Nº mínimo de iteraciones	saltar	49	50	1
		pasar 1 vez	49	51	2
		pasar 2 veces	49	52	3
		pasar m veces	49	100	51
		pasar n-1 veces	no hay nº máximo iteraciones		
		pasar n veces	no hay nº máximo iteraciones		
Bucle superior	saltar	Nº medio de iteraciones	50	100	50
	pasar 1 vez		49	100	51
	pasar 2 veces		48	100	52
	pasar m veces		0	100	100
	pasar n-1 veces		no hay nº máximo iteraciones		
	pasar n veces		no hay nº máximo iteraciones		

Bucles no estructurados

- Procedimiento
  - Este tipo de bucles son construcciones no deseables (ej., inclusión de instrucciones tipo *goto*) y lo que se debe de intentar en la medida de lo posible es el rediseñarlos para que se ajusten a las reglas de la programación estructurada

4.2 Pruebas de caja negra

- Algunas técnicas para definir casos de prueba
  - Particiones de equivalencia
  - Árboles/tablas de decisión
  - Análisis de valores límite
  - Valores típicos de error
  - Valores imposibles
- Con estas técnicas se diseñan casos de prueba tanto para condiciones de entrada inválidas o inesperadas como para condiciones válidas y esperadas

Técnica: Particiones de equivalencia

- El gran problema en la elaboración de casos de prueba es seleccionar el subconjunto de todas las entradas posibles al programa
- Las dos propiedades que deben cumplir estos subconjuntos son:
  - Reducir significativamente el número de los otros casos de prueba
    - minimizar el número total necesario de casos
  - Cubrir un conjunto extenso de otros casos de prueba posibles
    - partir el dominio de entrada de un programa en un número finito de clases de equivalencia
- La realización de casos de prueba por medio de particiones de equivalencia consta de dos pasos:
  - 1. Identificación de las clases de equivalencia
  - 2. Definición de los casos de prueba

Identificación de las clases de equivalencia

- Las clases de equivalencia se identifican tomando cada condición de entrada (generalmente una sentencia o frase en la especificación) y dividiéndola en dos o más grupos
  - por cada condición externa se establecen clases de equivalencia válidas (entradas válidas al programa) e inválidas (valores erróneos de entrada)
- Se pueden definir de acuerdo a las siguientes directrices:
  - Si una condición de entrada especifica un rango de valores (por ejemplo: valor entre 1 y 999), se definen una clase de equivalencia válida ($1 \leq valor \leq 999$) y dos inválidas (valor<1 y valor>999).
  - Si una condición de entrada requiere un valor específico, (el tamaño de un código es de 5 dígitos), se definen una clase de equivalencia válida (tamaño del código 5) y dos inválidas (tamaño<5 y tamaño>5).

Identificación de las clases de equivalencia (II)

- Si una condición de entrada especifica un conjunto de valores de entrada (tipo de vehículo: AUTOBÚS, TAXI, TURISMO, MOTOCICLETA), se define una clase de equivalencia válida para cada valor y una clase de equivalencia inválida. La clase de equivalencia inválida viene dada por cualquier elemento que no pertenezca al conjunto (por ejemplo PATINES).
- Si una condición de entrada es booleana (el primer carácter de un identificador debe ser una letra), se definen una clase válida (es letra) y una inválida (no es letra)

Entrada	Clases de Equivalencia Válidas	Clases de Equivalencia No Válidas

Definición de los casos de prueba

- El segundo paso es identificar los casos de prueba para cada clase de equivalencia, esto se realiza en los pasos siguientes:
  - Asignar un número único a cada clase de equivalencia
  - Hasta que todas las clases de equivalencia válidas hayan sido cubiertas por casos de prueba, escribir un nuevo caso de prueba que cubra tantas clases de equivalencia válidas no cubiertas como sea posible
  - Hasta que todas las clases de equivalencia inválidas hayan sido cubiertas por casos de prueba, escribir un caso de prueba para cubrir una y solo una de las clases de equivalencia inválida no cubierta hasta el momento

Ejemplo de casos de prueba con particiones de equivalencia (I)

- Dada la siguiente definición de una función, dar las clases de equivalencia y diseñar las pruebas de caja negra para verificar su correcto funcionamiento.
  - El perfil de la función es:

    ```
    int unaFuncion( int parametro1, string parametro2)
    ```

  - *parametro1* es un entero que toma valores entre 1 y 10, y entre 13 y 33
  - *parametro2* es una cadena de caracteres cuya longitud varía entre 3 y 10. Los caracteres pueden ser dígitos, letras minúsculas o los caracteres especiales "-" y "_"
  - Como resultado la función devuelve la suma del valor de *parametro1* y la longitud de *parametro2* (-1 si los parámetros de entrada no son correctos)

Ejemplo de casos de prueba con particiones de equivalencia (II)

- Identificación de clases de equivalencia

Entrada	Clases de equivalencia válidas	Clases de equivalencia no válidas
parametro1	**1)** 1 <= parametro1 <= 10 **2)** 13 <= parametro1<= 33	**5)** parametro 1 <1 **6)** 10 < parametro1 < 13 **7)** 33 < parametro1
parametro2	**3)** 3 <= length(parametro2) <= 10 **4)** parametro2 solo contiene caracteres incluidos en ['0'..'9','a'..'z','-',' ']	**8)** length(parametro2) < 3 **9)** length(parametro2) >10 **10)** parametro2 contiene otros caracteres

Ejemplo de casos de prueba con particiones de equivalencia (III)

- Definición de casos de prueba

Entrada		**Objetivo: clases cubiertas**	**Resultado**
parametro1	**parametro2**		
Casos de prueba para clases de equivalencia válidas (nº mínimo de casos de prueba)			
5	"09a_"	1,3,4	9
20	"09a_"	2,3,4	24
Casos de prueba para clases de equivalencia no válidas (uno por clase)			
0	"09a_"	5	-1
12	"09a_"	6	-1
35	"09a_"	7	-1
5	"ab"	8	-1
5	"abcdefghijklm"	9	-1
5	"09a?"	10	-1

Otro ejemplo de casos de prueba con particiones de equivalencia (I)

- Dada la siguiente definición de una función que realiza una búsqueda dicotómica en un vector de enteros, dar las clases de equivalencia y diseñar las pruebas de caja negra para verificar su correcto funcionamiento.
  - La cabecera de la función es:
    - `int busquedaDicotomica(int pVector[], int pClave)`
  - *pVector* tiene una capacidad máxima de 10 componentes. Cada componente del vector toma valores entre -500 y 0, y entre 500 y 1000. La ordenación de las componentes del vector es decreciente.
  - *pClave* es el valor del entero a buscar entre las componentes de *pVector* y que cumple las mismas restricciones que las componentes de *pVector*.
  - Si la búsqueda ha tenido éxito, la función devuelve el índice de la componente. En caso contrario, devuelve -1.

Otro ejemplo de casos de prueba con particiones de equivalencia (II)

Clases de equivalencia válidas

1. 0 ≤ capacidad(pVector) ≤ 10

2. las componentes de pVector toman valores entre -500 y 0

3. las componentes de pVector toman valores entre 500 y 1000

4. ∀ i, pVector[i] ≥ pVector[i+1]

5. –500 ≤ pClave ≤ 0

6. 500 ≤ pClave ≤ 1000

Clases de equivalencia no válidas

7. capacidad(pVector) > 10

8. ∃ i, pVector[i] < –500

9. ∃ i, 0 < pVector[i] < 500

10. ∃ i, pVector[i] > 1000

11. ∃ i, pVector[i] < pVector[i+1]

12. pClave < –500

13. 0 < pClave < 500

14. pClave > 1000

Otro ejemplo de casos de prueba con particiones de equivalencia (III)

- Casos de prueba válidos (nº mínimo de casos de prueba)

Entrada	**Resultado esperado**	**Clases equiv. cubiertas**
pVector={500, 0, –1, –2} pClave = –3	-1	1, 2, 3, 4, 5
pVector={500, 0, –1, –2} pClave = 500	0	1, 2, 3, 4, 6

Ejercicio de pruebas de caja negra

- Casos de prueba no válidos (uno por clase)

Entrada	**Resultado esperado**	**Clase equiv.cubierta**
pVector={0,–1,–2,–3,–4,–5,–6,–7,–8,–9,–10} pClave = –3	-1	7
pVector={500, 0, –1, –2, –1000} pClave = 500	-1	8
pVector={500, 250, 0, –1, –2} pClave = 500	-1	9
pVector={1001, 500, 0, –1, –2} pClave = 750	-1	10
pVector={500, 501, –1, –2} pClave = 501	-1	11
pVector={0,–1,–2,–3,–4} pClave = –750	-1	12
pVector={0,–1,–2,–3,–4} pClave = 250	-1	13
pVector={500, 0, –1, –2} pClave = 1001	-1	14

Técnica: Árboles/tablas de decisión

- La técnica de árboles y tablas de decisión para evitar la ambigüedad de la especificación de requisitos en la fase de requisitos también se puede utilizar para definir casos de prueba
- Las condiciones de entrada y las acciones a realizar permiten diseñar casos de prueba
  - Condiciones de entrada -> Valores de entrada
  - Acciones a realizar (reglas de decisión) -> Resultados esperados

Ejemplo de casos de prueba con tablas de decisión

- Especificación de la funcionalidad del sistema:
  - Si la cuenta del cliente se factura usando un método de tarificación fijo, se establece una carga mensual mínima para consumos menores de 100 Kwh. En los demás casos, la facturación aplica la tarifa A
  - Sin embargo, si la cuenta se factura usando un método de facturación variable, se aplicará la tarifa A a los consumos menores de 100Kwh. En otro caso, se factura de acuerdo a la tarifa B
- Tabla de decisión

C1:Cuenta de tarifa fija	S	S	N	N
C2: Consumo < 100 Kwh	S	N	S	N
A1: Cargo mensual mínimo	X			
A2: Tarifa A		X	X	
A3: Tarifa B				X

- Casos de prueba

ID	**Entrada**		**Resultado esperado**
	Método tarifación	**Consumo**	**Coste**
1	Fijo	50 Kwh	Cargo mensual mínimo
2	Fijo	150 Kwh	150 * tarifa A
3	Variable	50 Kwh	50 * tarifa A
4	Variable	150 Kwh	150 * tarifa B

Técnica: Análisis de valores límite

- La técnica de análisis de valores límite selecciona casos de prueba que ejerciten los valores límite dada la tendencia de la aglomeración de errores en los extremos
- Complementa la prueba de partición equivalente y presenta notables diferencias:
  - En lugar de realizar la prueba con único elemento representativo de la partición equivalente, se escogen uno o más valores de forma que cada margen de la clase de equivalencia es objeto de una prueba

Análisis de valores límite (II)

- Cómo derivar casos de prueba:
  - Si una condición de entrada especifica un *rango* delimitado por los valores *a* y *b*, se deben diseñar casos de prueba para los valores *a* y *b* y para los valores justo por debajo y justo por encima de ambos.
    - Por ejemplo, si una variable toma valores entre 20 y 10.000, se deben escoger los siguientes valores entre los casos de prueba:
      - 19, 20
      - 10.000, 10.001
    - Si los recursos lo permiten, se podrían añadir pruebas adicionales:
      - Un valor cualquiera de la mitad del rango
      - Valores negativos, al menos un valor positivo menor que 19, algún valor muy superior a 10.000

Análisis de valores límite (III)

- Si la condición de entrada especifica un *número* de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo además de los valores justo encima y debajo de aquéllos.
  - Ejemplo: si el número de alumnos para calcular su nota final es 125, escribir casos de prueba para 0,1,125 y 126
- Si las estructuras de datos definidas internamente tienen límites prefijados (ej: un array de 10 entradas), se debe asegurar diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

- Normalmente, esta técnica se lleva a cabo de manera intuitiva cuando se prueba software, aunque no siempre se ejerciten todos los casos que aquí se exponen

Ejemplo de análisis de los valores límite (I)

- Supongamos una aplicación de cálculo de la nómina de una empresa de 132 empleados.
  - La información necesaria para calcular el salario de cada empleado es :
    - Código : entero de 1 a 132
    - Puesto : alfanumérico de hasta 4 caracteres
    - Antigüedad : real de 0 a 25 años
    - Horas semanales : 0 a 60
  - La aplicación devuelve un valor real >=0.0 siempre que las entradas sean correctas, o negativo en caso contrario

Condición de entrada	**Valores límite a seleccionar**
Código	0; 1; 132; 133
Longitud tipo puesto	0; 1; 4; 5
Antigüedad	-0.1; 0; 25; 25.1
Horas semanales	-1; 0; 60; 61

Ejemplo de análisis de los valores límite (II)

- Casos de prueba
  - Por cada condición de entrada se generan casos de prueba con cada uno de los posibles valores manteniendo el resto de condiciones de entrada con un valor válido arbitrario

ID	Entradas				Resultado esperado
	Código	Puesto	Antigüedad	Horas	
1	0	"A"	10	40	Valor real < 0.0
2	1	"A"	10	40	Valor real >= 0.0
3	132	"A"	10	40	Valor real >= 0.0
4	133	"A"	10	40	Valor real < 0.0
5	20	""	10	40	Valor real < 0.0
6	…				

Técnica: Valores típicos de error

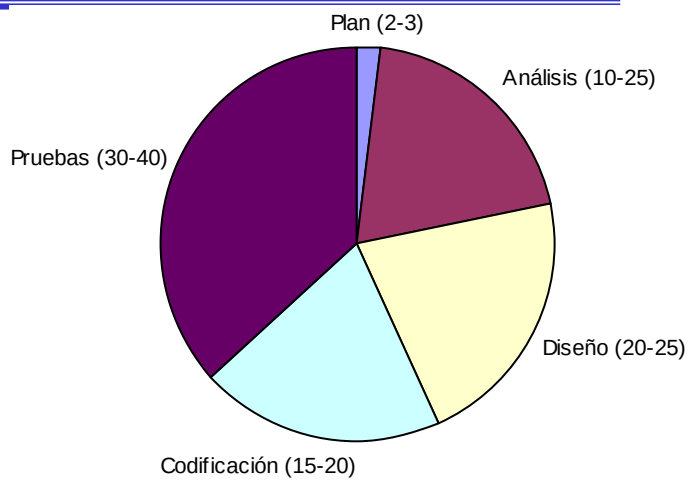
- Ciertos valores son una fuente de error bastante común, así que se deben especificar entre los datos de prueba
- La naturaleza y funcionalidad del sistema a probar determinan cuáles son los valores susceptibles de causar problemas
- Ejemplo: campos identificativos nulos

Técnica: Valores imposibles

- Son aquellos valores de datos que, dentro del sistema a probar, han sido especificados como no posibles y que, sin embargo, pueden haber sido generados internamente por el sistema, provocando como consecuencia el mal funcionamiento o incluso la parada del sistema.
- Es conveniente incluirlos como casos de prueba a determinar:
  - Si pueden ser detectados
  - Si el sistema puede manejarlos adecuadamente sin provocar errores irreparables
- Ejemplo: ficheros de datos vacíos, valores de configuración erróneos

5. Automatización de las pruebas

- Las pruebas pueden suponer hasta el **40%** del tiempo de desarrollo
- Las pruebas son polémicas:
  - No son parte de la solución → no se entregan a los clientes (pueden dar incluso mala impresión)
  - Son difíciles de mantener
- ¿Qué se puede hacer?
  - Intentar automatizarlas mediante la utilización de **herramientas**
- ¿Qué se puede automatizar?
  - Diseño de las pruebas
  - Codificación de las pruebas
  - Ejecución de las pruebas
  - Evaluación de resultados



⇨ Menos habitual, pocas técnicas y herramientas

⇨ Lo más habitual, muchas técnicas y herramientas

Categorías de herramientas

- Analizadores estáticos
  - Evaluar el software sin ejecutarlo
- Auditores de código
  - Filtros que verifican que el código fuente cumple ciertos criterios de calidad
- Generadores de archivos y datos de prueba
- Controladores de prueba
  - Ayudan a la realización y repetición de pruebas
  - Simulan comportamientos
- Simuladores
  - En muchas ocasiones el sistema no se puede probar en entornos reales, en estos casos es necesario simular el entorno
  - Suelen integrar generadores de datos y controladores de prueba
  - Útiles para distintos tipos de prueba: unidad, sistema
- Herramientas de monitorización de recursos (memoria, CPU, …)

Controladores de pruebas

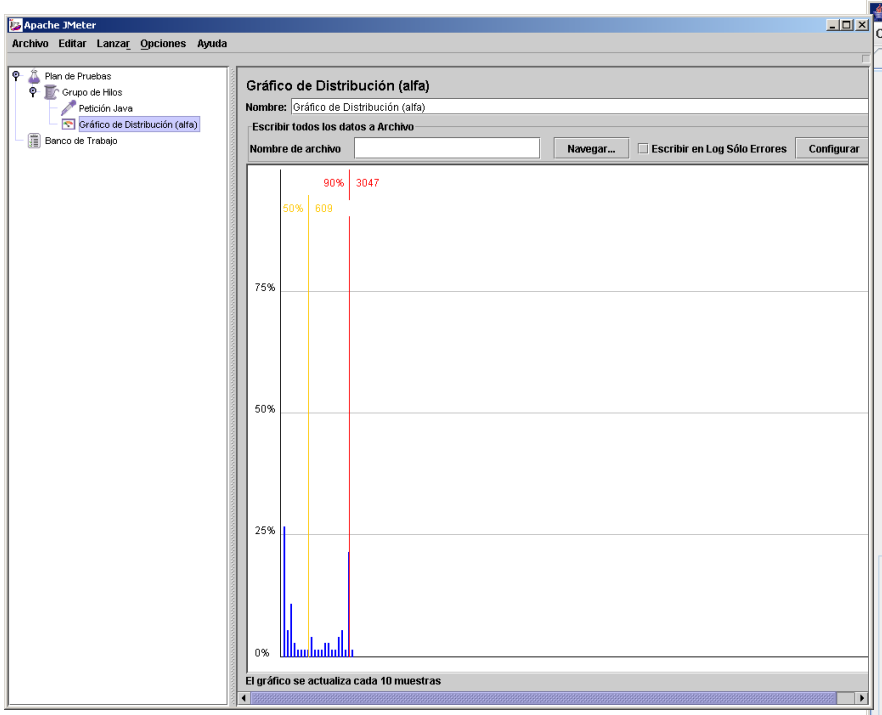
- Algunas herramientas: JUnit (programas Java), Jakarta Cactus (servlets Java), JWebUnit (aplicaciones Web)
- **JUnit**
  - Es un marco para escribir **tests repetibles**
  - No se requiere interpretación humana para interpretar las pruebas
  - Modo de trabajo para probar un método
    - **Anotar** un método con @org.junit.Test
    - Cuando se quiere comprobar un valor, hacer una llamada al método **assertTrue** y pasar una expresión booleana que tome el valor verdad si el test es satisfactorio
  - Ejemplo:

```
@Test public void simpleAdd() {
  Money m12CHF = new Money(12, "CHF");
  Money m14CHF = new Money(14, "CHF");
  Money expected = new Money(26, "CHF");
  Money result = m12CHF.add(m14CHF);
  assertTrue(expected.equals(result));
}
```

Herramientas de monitorización de recursos

- Algunas herramientas: JMeter, JConsole

JMeter, tiempos de ejecución de cada hilo



JConsole, utilización de memoria