INGENIERÍA DEL SOFTWARE II

Tema 1

Construcción y Pruebas de Software

Univ. Cantabria – Fac. de Ciencias

Carlos Blanco

Carlos.Blanco@unican.es

Objetivos

- Construcción del Software
  - Comprender que **construir software** engloba muchas **más** actividades que la de **escribir código**
  - Conocer los **principios de la construcción** de software y las **actividades** más significativas
- Verificación y Validación
  - Conocer el papel que juegan la **verificación y validación** del software y, dentro de ellas, las **pruebas**
- Pruebas
  - Tener una visión general de los **niveles, clases, técnicas y actividades de pruebas** del software
- Pruebas OO
  - Aprender las principales **estrategias y métodos** de pruebas para sistemas OO
  - Aprender a **diseñar** casos de pruebas para OO

Contenido

1. **Construcción del Software**
   - Conceptos
   - Principios
   - Proceso de Construcción
2. **Verificación y Validación**
   - Objetivos
   - Actividades
   - Técnicas
3. **Pruebas**
   - Conceptos
   - Proceso de Pruebas
   - Niveles de Prueba
   - Estrategia de Aplicación
   - Técnicas de Prueba
4. **Pruebas de Sistemas OO**
   - Introducción
   - Estrategias para sistemas OO
   - Diseño de casos de prueba OO
   - Métodos de pruebas OO

Bibliografía

- Bibliografía
  - Piattini (2007). (Cap. 10)
  - Sommerville (2005). (Capítulo 22 y 23)
  - Jacobson, I., Booch, G., and Rumbaugh, J. (2000): El Proceso Unificado de Desarrollo. Addison-Wesley. (Capítulo 11)
  - Pressman, R. (2005): Ingeniería del Software: Un Enfoque Práctico. 6º Edición. McGraw-Hill. (Capítulos 13 y 14)
  - Pfleeger (2002). (Caps. 7, 8 y 9)
  - IEEE Computer Society (2004). SWEBOK - Guide to the Software Engineering Body of Knowledge, 2004. (Capítulos 4 y 5) http://www.swebok.org/

1. Construcción

- Se refiere a la creación detallada de **software operativo** mediante una combinación de
  - Codificación
  - Verificación
  - Pruebas Unitarias y de Integración
  - Depuración

1.Construcción

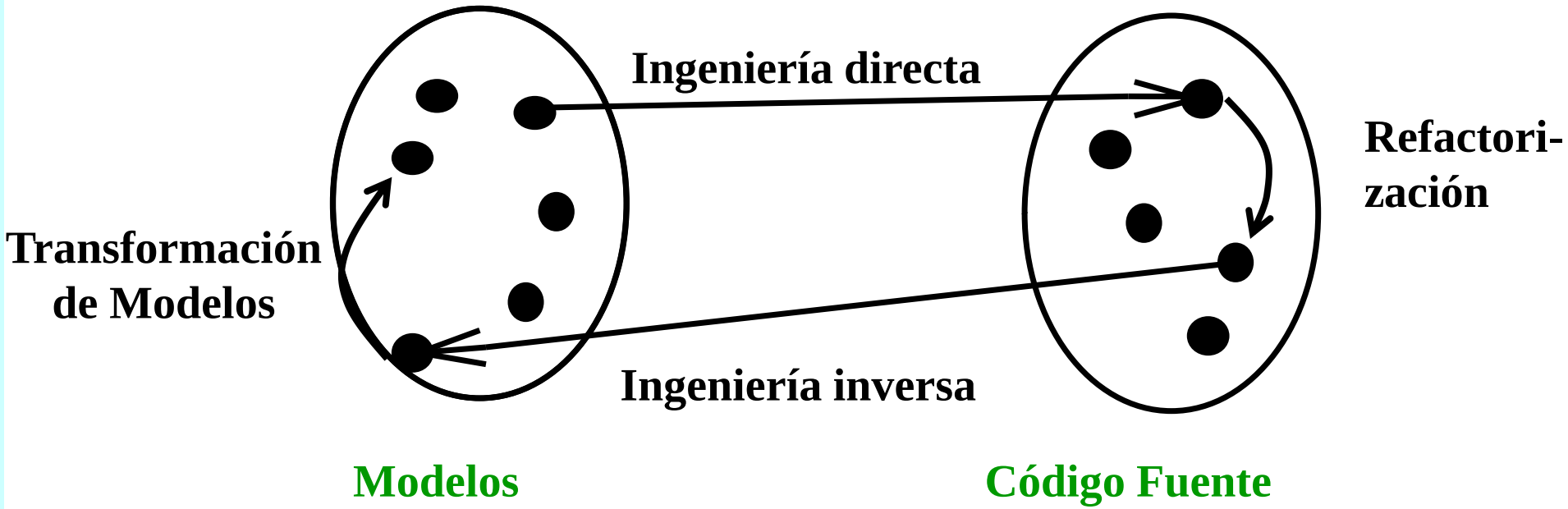
- Los límites entre Construcción, Diseño y Pruebas varían dependiendo del **ciclo de vida** que se usa en cada proyecto
  - Aunque bastante esfuerzo de **diseño** puede realizarse antes de empezar las actividades de construcción, otro debe de realizarse en paralelo
  - Igualmente, algunos tipos de **pruebas** se realizan durante la construcción, mientras que otras se hacen a posteriori
- Durante la Construcción se generan las cantidades más grandes de artefactos software (ficheros de código, contenidos, casos de prueba, etc.)
  - Esto origina necesidades de **gestión de configuración**

1.Construcción

- **Lenguajes de Construcción**
  - Formas de especificar una solución a un problema ejecutable por una computadora
  - Pueden ser de varias **clases**:
    - De Configuración
      - Permiten elegir entre un conjunto predefinido de opciones para crear instalaciones de software nuevas o particularizadas (ej. ficheros de configuración de Windows o UNIX)
    - De Toolkits
      - Permiten construir aplicaciones a partir de un Toolkit (conjunto integrado piezas reutilizables de aplicación específica)
        - **Scripts**: definidos de forma explícita. (ej. JavaScript).
        - **APIs**: definidos de forma implícita, ya que se implican a partir de la interfaz pública de un Toolkit. (ej. API de Office).
    - De Programación
      - Tipos de notaciones: Lingüística (cadenas de texto con palabras), Formal (expresiones de tipo matemático), Visual (símbolos gráficos)

1.Construcción

- Relación de los **modelos** con el **código fuente**



1.Construcción - Principios

- Los **principios fundamentales** de la construcción de software son:
  - Minimizar la Complejidad
  - Anticipar los Cambios
  - Pensar en la Verificación posterior
  - Aplicar Estándares

1.Construcción - Principios

- Los **principios fundamentales** de la construcción de software son:
  - Minimizar la Complejidad
    - Escribiendo **código sencillo y fácil de leer**
    - Utilizando estándares
    - Técnicas de codificación
    - Técnicas de aseguramiento de calidad
  - Anticipar los Cambios
    - El software se ve afectado por los cambios en su entorno y está destinado a cambiar a lo largo del tiempo
    - Aplicación de técnicas específicas

1.Construcción - Principios

- Los **principios fundamentales** de la construcción de software son:
  - Pensar en la Verificación posterior
    - Construir de forma que los fallos puedan ser encontrados lo antes posible (al codificar, hacer pruebas u operar el sistema)
    - Técnicas:
      - Seguir estándares de codificación
      - Hacer pruebas unitarias
      - Organizar el código para soportar pruebas automatizadas
      - Restringir el uso de técnicas complejas

1.Construcción - Principios

- Los **principios fundamentales** de la construcción de software son:
  - Aplicar Estándares
    - Directamente a la construcción del software:
      - Formatos de comunicación (documentos, contenidos).
      - Versiones estándares de lenguajes de programación (Java, C++, C#,…).
      - Reglas de codificación (nombres de variables, comentarios,…).
      - Notaciones de diagramas (UML,…).
      - Intercambio entre herramientas (XLM,…).
    - En un proyecto:
      - **Externos**: propuestos por organismos de estandarización (ISO, ANSI, AENOR), consorcios industriales (OMG), asociaciones profesionales (IEEE, ACM).
      - **Internos**: creados por la propia organización.

1.Construcción - Proceso

- Desde una **perspectiva de proceso**,

  los esfuerzos más significativos que se realizan durante la construcción de software son:

  - Planificación
  - Manejo de Excepciones
  - Codificación
  - Pruebas
  - Aseguramiento de Calidad
  - Reutilización
  - Integración

1.Construcción - Proceso

- **Planificación**
  - Elegir el método de construcción
    - Granularidad y alcance con el que se realizan los requisitos
    - Orden en el que se abordan
  - Establecer el orden en el que los componentes se crean e integran
  - Asignar tareas de construcción a personas específicas
- **Manejo de Excepciones**
  - Mientras se construye un edificio, los albañiles deben hacer pequeños ajustes respecto de los planos
  - De igual forma, los constructores de software deben hacer modificaciones y ajustes, unas veces pequeñas y otras no tanto, respecto de los detalles del diseño de un software

1.Construcción - Proceso

- **Codificación**
  - La escritura del código fuente es el principal esfuerzo de construcción de software
  - Aplicar técnicas para crear código fuente comprensible (reglas de asignación de nombres y de formato del código, clases, tipos enumerados, constantes etiquetadas,…)
  - Manejar condiciones de error (errores previstos e imprevistos, excepciones)
  - Prevenir brechas de seguridad a nivel de código (llenado de buffers, overflow de índices de vectores, …)
  - Uso eficiente de recursos escasos (hilos, bloqueos en bases de datos, …)
  - Organizar el código fuente (sentencias, rutinas, clases, paquetes, …)
  - Documentar el código

1.Construcción - Proceso

- **Pruebas**
  - Frecuentemente realizadas por los mismos que escriben el código
  - El propósito de estas pruebas es reducir el tiempo entre el momento en el que los fallos se insertan en el código y el momento en que son detectados.
    - **Pruebas Unitarias**
    - **Pruebas de Integración**
- Técnicas para **asegurar la calidad** del código
  - Pruebas (Unitarias y de Integración)
  - Escribir las pruebas primero (*test first development*)
  - Ejecución línea a línea (*code stepping*)
  - Uso de aserciones
  - Depuración (*debugging*)
  - Revisiones
  - Análisis estático

1.Construcción - Proceso

- **Reutilización**
  - Implica más cosas que crear y usar bibliotecas de activos
  - Es necesario oficializar la práctica de reutilización integrándola en los ciclos de vida y metodologías de los proyectos
  - Activos reutilizables:
    - Planes de proyecto, Estimaciones de coste
    - Especificaciones de requisitos, Arquitecturas, Diseños, Interfaces…
    - Código fuente (librerías, componentes,…)
    - Documentación de usuario y técnica, Casos de prueba, Datos,…
- **Integración**
  - De rutinas, clases, componentes y sub-sistemas construidos de forma separada, y con otro(s) sistema(s)
  - Para ello puede ser necesario:
    - Planificar la secuencia en que se integrarán los componentes
    - Crear "andamios" para soportar versiones provisionales
    - Determinar el nivel de pruebas y aseguramiento de calidad realizado sobre los componentes antes de su integración

2. Verificación y Validación

- **VV** es un conjunto de procedimientos, actividades, técnicas y herramientas que se utilizan, paralelamente al desarrollo de software, para asegurar que un producto software resuelve el problema inicialmente planteado
- Las pruebas son una familia de técnicas de **VV**

2.Verificación y Validación

- **Objetivos de la VV**:

  Actúa sobre los productos intermedios que se generan durante el desarrollo para:

  - **Detectar y corregir cuanto antes** sus defectos y las desviaciones respecto al objetivo fijado
  - Disminuir los **riesgos**, las desviaciones sobre los presupuestos y sobre el calendario o programa de tiempos del proyecto
  - Mejorar la **calidad y fiabilidad** del software
  - **Valorar** rápidamente los **cambios** propuestos y sus consecuencias

2.Verificación y Validación

- La visión del desarrollo de software como un conjunto de **fases** con posibles realimentaciones facilita la **VV**
  - Al inicio del proyecto es necesario hacer un

    Plan de VV del Software

    (estructura del plan según el estándar IEEE 1012)
  - Las actividades de VV se realizan de forma **iterativa** durante el desarrollo

IEEE 1012-2004: IEEE Standard for Software Verification and Validation

1. Propósito
2. Documentos de referencia
3. Definiciones
4. Visión general de la verificación y validación
4.1 Organización
4.2 Programa de tiempos
4.3 Esquema de integridad de software
4.4 Resumen de recursos
4.5 Responsabilidades
4.6 Herramientas, técnicas y metodologías
5. Verificación y validación en el ciclo de vida
5.1 Gestión de la VV
5.2 VV en el proceso de adquisición
5.3 VV en el proceso de suministro
5.4 VV en el proceso de desarrollo:
    5.4.1 VV de la fase de concepto
    5.4.2 VV de la fase de requisitos
    5.4.3 VV de la fase de diseño
    5.4.4 VV de la fase de implementación
    5.4.5 VV de la fase de pruebas
    5.4.6 VV de la fase de instalación
5.5 VV de la fase de operación
5.6 VV del mantenimiento
6. Informes de la VV del software
7. Procedimientos administrativos de la VV
7.1 Informe y resolución de anomalías
7.2 Política de iteración de tareas
7.3 Política de desviación
7.4 Procedimientos de control
7.5 Estándares, prácticas y convenciones.
8. Requisitos de documentación para la VV

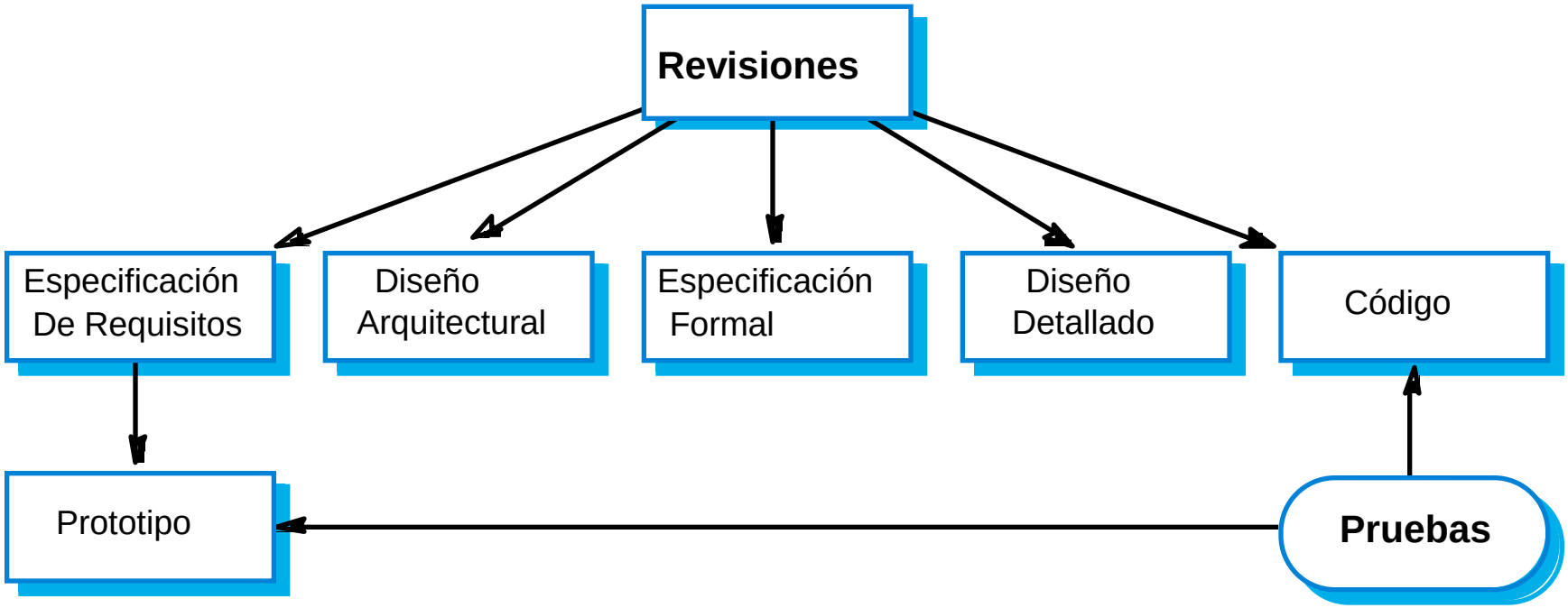
2.Verificación y Validación

- **Actividades de VV**

	VERIFICACIÓN	VALIDACIÓN
	¿Estamos **construyendo correctamente** el producto?	¿Estamos construyendo el **producto correcto**?
El software debe	Estar conforme a su especificación	Hacer lo que el usuario realmente quiere
Objetivo	Demostrar la consistencia, compleción y corrección de los **artefactos** de las distintas fases (productos intermedios)	Determinar la corrección del **producto final** respecto a las necesidades del usuario
Técnica más utilizada	Revisiones	Pruebas

2.Verificación y Validación

- **Técnicas** Estáticas (Revisiones) vs Dinámicas (Pruebas)



2.Verificación y Validación

- Las **revisiones** software pueden ser
  - Informales
    - No hay procedimientos definidos, por lo que la revisión se realiza de la forma más flexible posible.
    - Ventajas → menor coste y esfuerzo, preparación corta, etc.
    - Desventajas → Detectan menos defectos
  - Semi-formales
    - Se definen unos procedimientos mínimos a seguir (*walkthroughs)*
  - Formales (**Inspecciones**)
    - Se define completamente el proceso
    - Revisión en detalle, por una persona o grupo distintos del autor, para:
      - Verificar si el producto se ajusta a sus **especificaciones** o **atributos de calidad** y a los **estándares** utilizados en la empresa
      - Señalar las **desviaciones** sobre los estándares y las especificaciones
      - **Recopilar datos** que realimenten inspecciones posteriores (defectos recogidos, esfuerzo empleado, etc.)

2.Verificación y Validación

- Informes de Inspección
  - Ejemplo de Formulario

Informe de inspección

Proyecto
Fecha de revisión **Elemento revisado**
Documento **Versión**
Moderador
Revisores

Tipo de reunión: [ ] Revisión [ ] Re-revisión [ ] Mantenimiento

Tipo de inspección:

☐ Requisitos (ARS) ☐ Especificación (EFS) ☐ Arquitectura (DTS)

☐ Diseño (DTS) ☐ Código (DCS) ☐ Diseño de Pruebas (DTS)

☐ Casos de prueba (DCS)

Decisión: [ ] Aceptar [ ] Aceptar condicionalmente [ ] Rechazar

Tipo	**Defectos graves**				**Defectos leves**			
	Omisión	Error	Añadido	Total	Omisión	Error	Añadido	Total
Sintaxis								
Compleción								
Nombres								
Consistencia								

Esfuerzo:

- Preparación horas·persona
- Reuniones horas·persona
- Seguimiento horas-persona

Firma:

Moderador
D/Dña. _____

Secretario
D/Dña. _____

- **3. Pruebas**

- La manera adecuada de enfrentarse al reto de la **calidad** es la **prevención**
  - ***! Mas vale prevenir que curar !***

- Las pruebas son técnicas de **comprobación dinámica**
  - Siempre implican la **ejecución** del programa
  - Permiten:
    - Evaluar la calidad de un producto
    - Mejorarlo identificando defectos y problemas

3.Pruebas - Conceptos

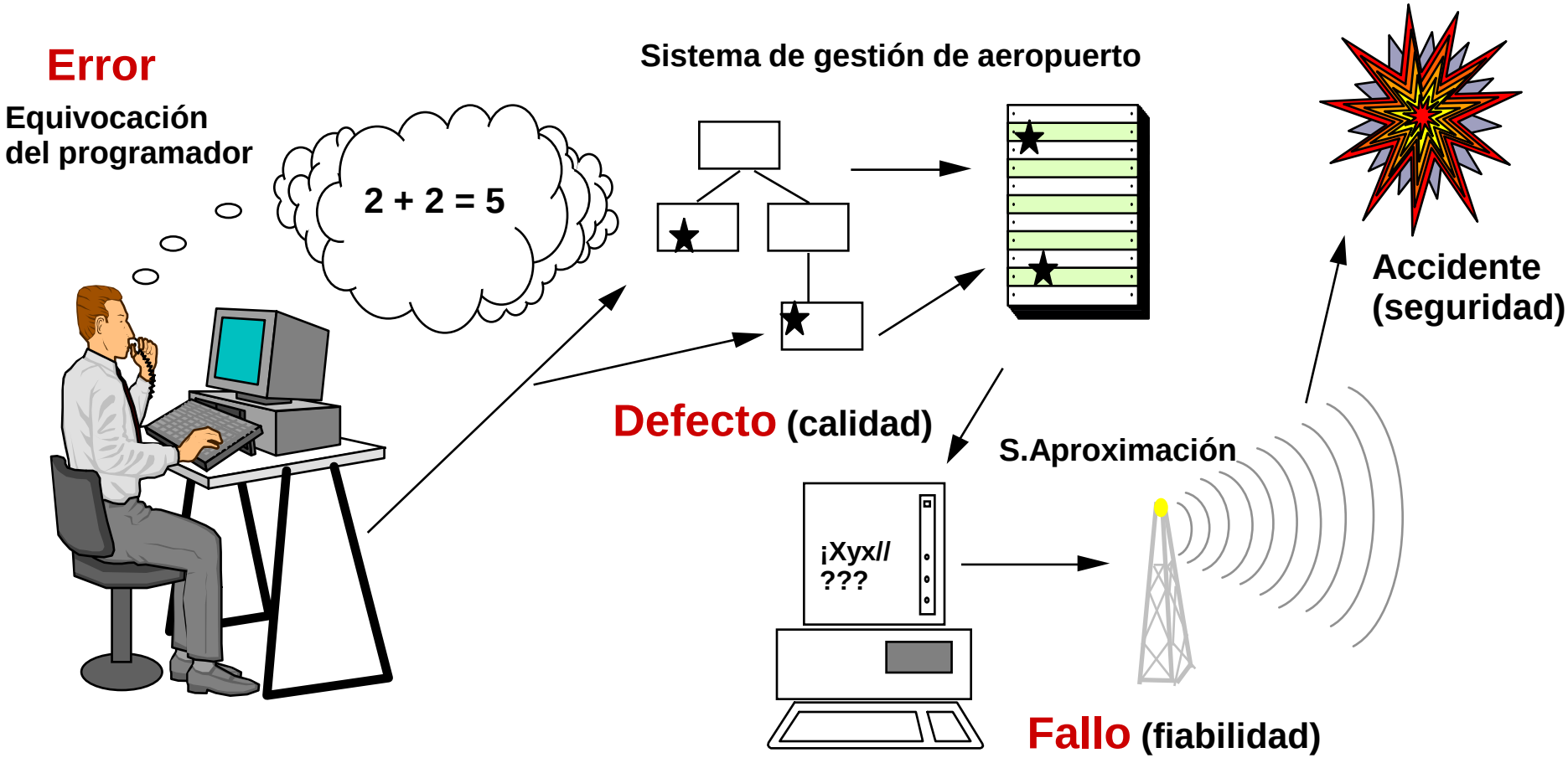
- **Ideas** preconcebidas sobre las Pruebas:
  - ¿Pueden detectar las pruebas la **ausencia** de errores?
  - ¿Se puede realizar una **prueba exhaustiva** del software (probar todas las posibilidades de su funcionamiento)?
  - ¿Cuándo se descubre un error la prueba ha tenido éxito o ha fracasado?

3.Pruebas - Conceptos

- Limitaciones Teóricas y Prácticas de las Pruebas
  - Aforismo de **Dijkstra**: "*Probar programas sirva para demostrar la presencia de errores, pero nunca para demostrar su ausencia".*
  - En el mundo real no es posible hacer pruebas **completas**
    - Se considera que existen infinitos casos de prueba y hay que buscar un equilibrio (recursos y tiempo limitados)
- Selección de los valores de entrada
  - Selección de un **conjunto finito** de casos de prueba
    - El criterio de selección depende de la técnica de pruebas
    - No siempre son suficientes, ya que un sistema complejo podría reaccionar de distinta manera ante una misma entrada dependiendo de su estado
- Comparación de la salida obtenida con la esperada
  - Decidir si los resultados observados son aceptables o no
    - Según expectativas de los usuarios, especificaciones, requisitos,…
- Oráculo
  - Agente (humano o no) que decide cuando un programa actúa correctamente en una prueba dada, emitiendo un veredicto de "correcto" o "fallo"

3.Pruebas - Conceptos

- Relación entre Error, Defecto y Fallo:



3.Pruebas - Conceptos

- **Ejemplo** de las dificultades que se presentan

```
if (A+B+C)/3 == A
then print (“A, B y C son iguales”)
else print (“A, B y C no son iguales”)
```

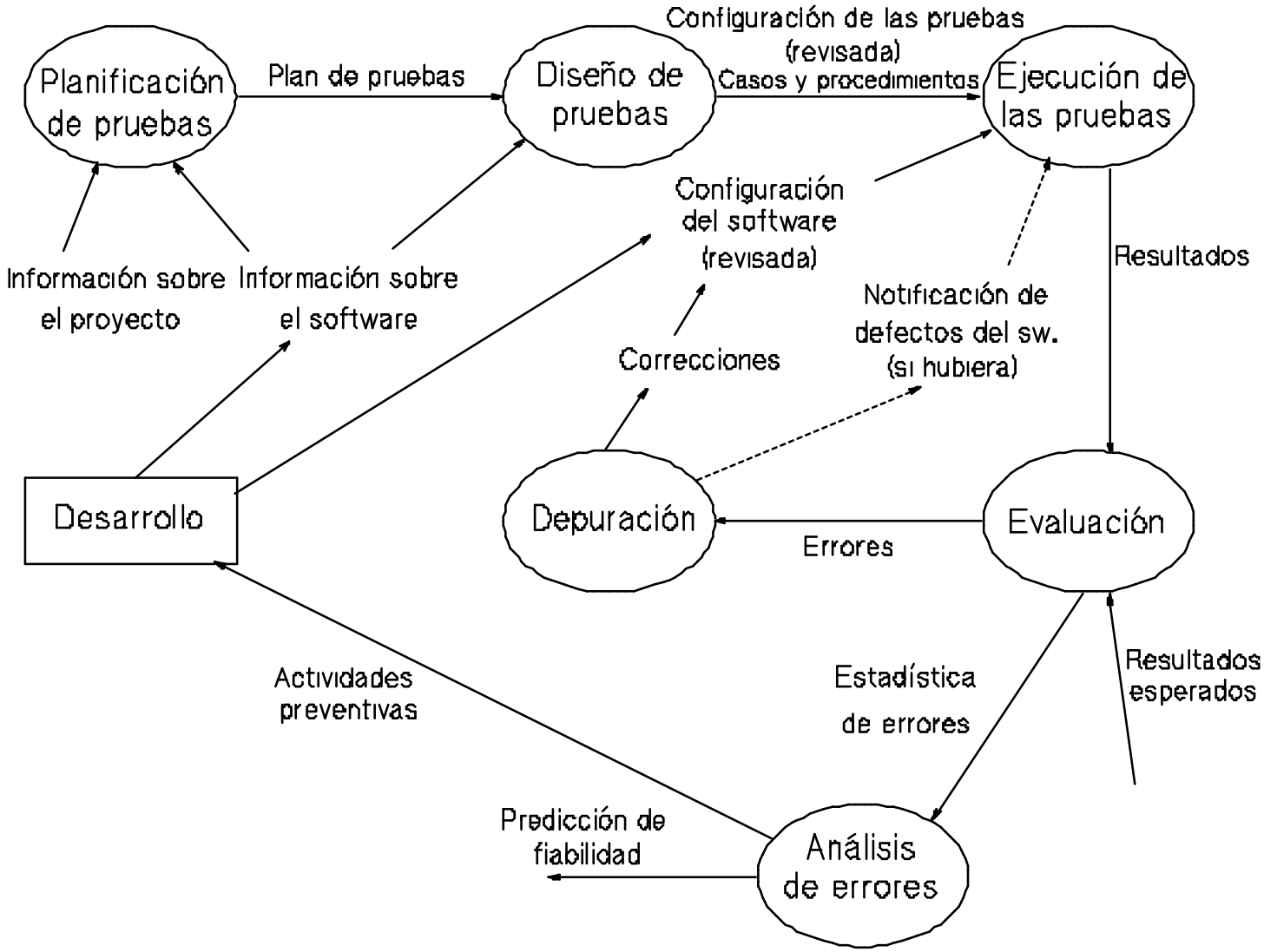
¿Son suficientes estos dos casos de prueba?

Caso 1: A=5; B=5; C=5;

Caso 2: A=2; B=3; C=7;

¿Cuáles otros serían necesarios en caso negativo?

3.Pruebas - Proceso



3.Pruebas - Niveles

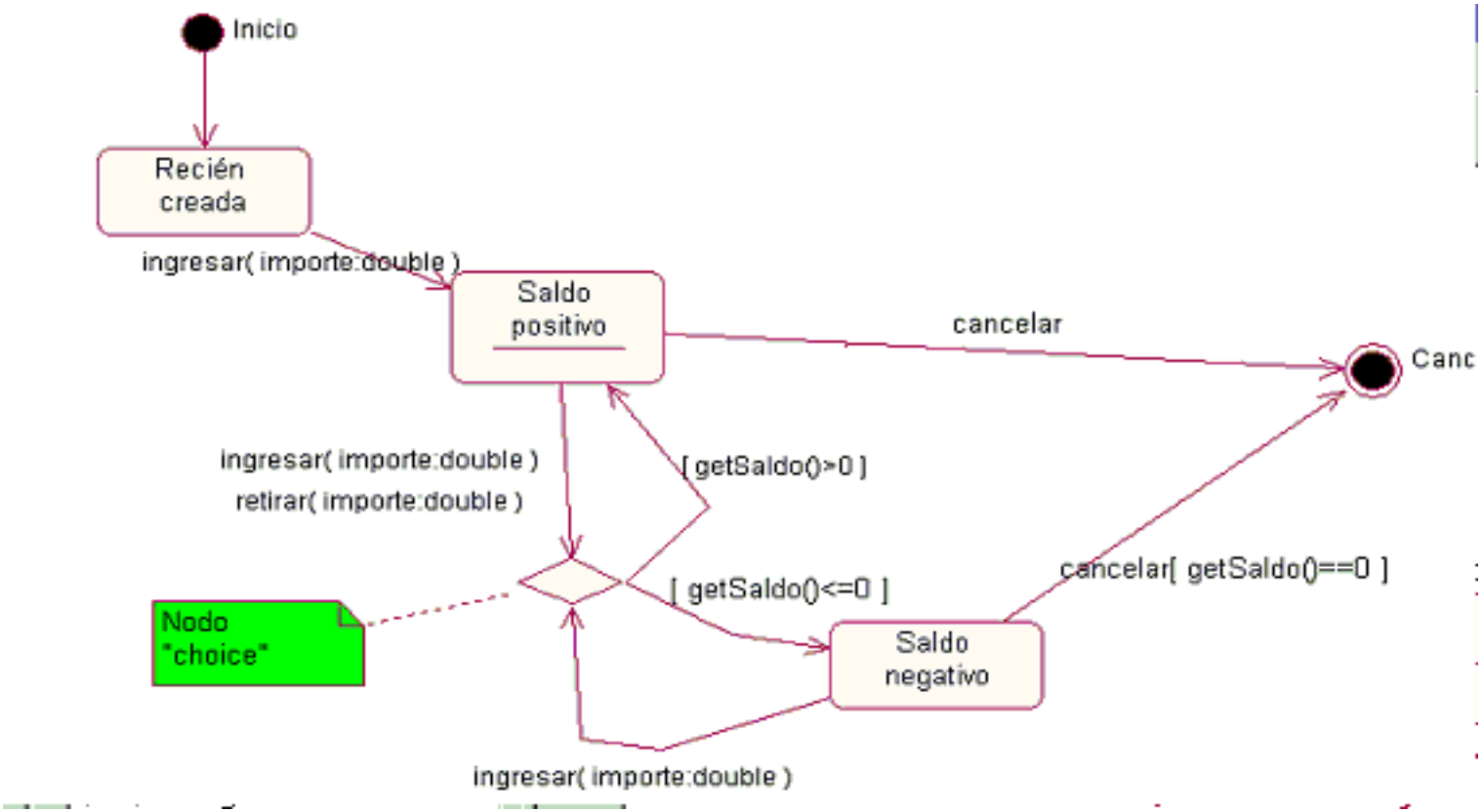
- El ámbito o destino de las pruebas del software puede variar en **tres niveles**:
  - Un **módulo único**
    - PRUEBAS UNITARIAS
  - Un **grupo de módulos** (relacionados por propósito, uso, comportamiento o estructura)
    - PRUEBAS DE INTEGRACIÓN
  - Un **sistema completo**
    - PRUEBAS DE SISTEMA

3.Pruebas - Niveles

- **Pruebas Unitarias**
  - Verifican el **funcionamiento aislado** de piezas de software que pueden ser probadas de forma separada
    - Subprogramas/Módulos individuales
    - Componente que incluye varios subprogramas/módulos
  - Estas pruebas suelen llevarse a cabo con:
    - Acceso al código fuente probado
    - Ayuda de herramientas de depuración
    - Participación (opcional) de los programadores que escribieron el código

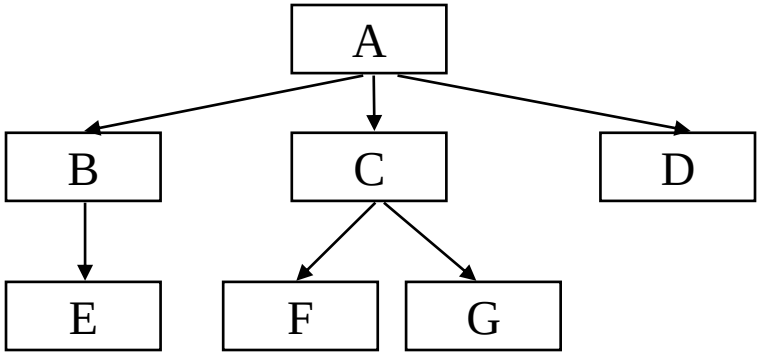
3.Pruebas - Niveles

- **Pruebas Unitarias**
  - ¿De dónde obtener **buenos casos de prueba unitarias**?
    - De las **máquinas** o diagramas **de estado**



- **Pruebas de Integración**
  - Verifican la **interacción entre componentes** del sistema software
  - Estrategias:
    - Guiadas por la arquitectura
      - Los componentes se integran según hilos de funcionalidad
    - Incremental
      - Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados

  - Incremental Ascendente (Bottom-Up)
    1. Se comienza por los módulos hoja (pruebas unitarias)
    2. Se combinan los módulos según la jerarquía
    3. Se repite en niveles superiores
  - Incremental Descendente (Top-Down)
    - Primero en profundidad, completando ramas del árbol
    - Primero en anchura, completando niveles de jerarquía



3.Pruebas - Niveles

- **Pruebas de Sistema**
  - Verifican el **comportamiento del sistema** en su conjunto
  - Los fallos **funcionales** se suelen detectar en los otros dos niveles anteriores (unitarias e integración)
  - Este nivel es más adecuado para comprobar **requisitos no funcionales**
    - Seguridad, Velocidad, Exactitud, Fiabilidad
  - También se prueban:
    - Interfaces externos con otros sistemas
    - Utilidades
    - Unidades físicas
    - Entorno operativo

3.Pruebas – Clasificación según finalidad

- Clasificación de pruebas según su **finalidad**:
  - Comprueban que las **especificaciones funcionales** están implementadas correctamente
    - Pruebas Unitarias y de Integración
  - Comprueban los **requisitos no funcionales**
    - Pruebas de Sistema
  - Comprueban el comportamiento del sistema frente a los **requisitos del cliente** (suele participar el mismo cliente o los usuarios)
    - Pruebas de Aceptación
  - Comprueban el comportamiento del sistema frente a los requisitos de **configuración hardware**
    - Pruebas de Instalación
  - Pruebas en grupos pequeños de **usuarios potenciales** antes de la difusión del software
    - Pruebas alfa (en la misma empresa) y beta (fuera)

3.Pruebas – Estrategia de Aplicación

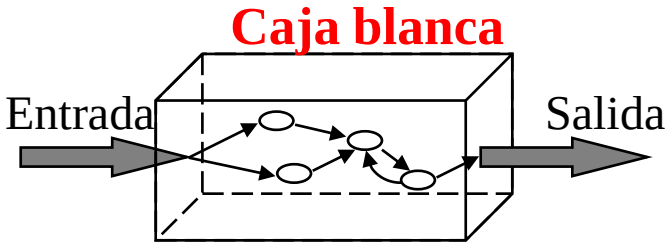
- **Relación entre las Actividades de Desarrollo y las Pruebas**
  - **¿ En qué orden han de escribirse y de realizarse las actividades de pruebas ?**

3.Pruebas – Técnicas

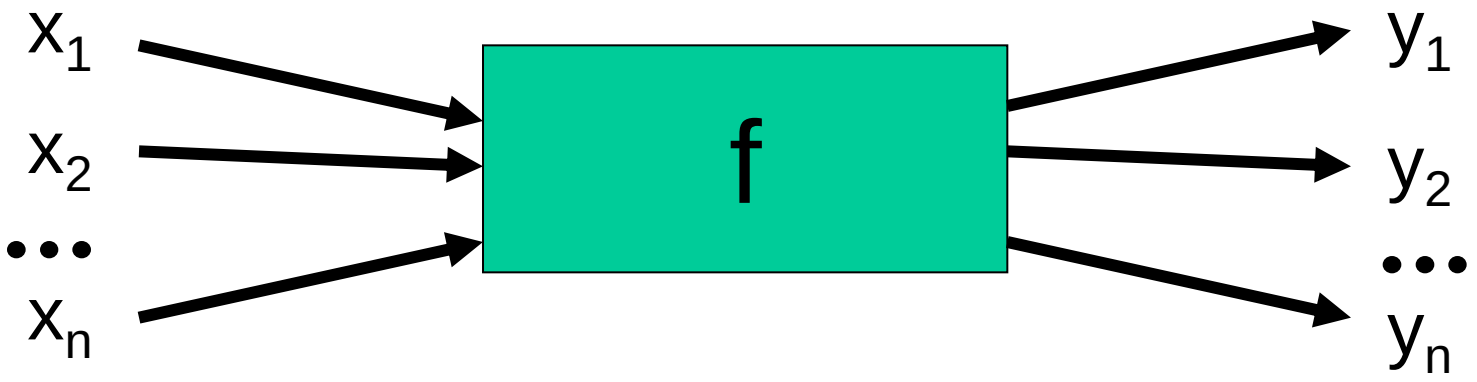
- **Técnicas de Prueba**
  - **Principio básico:**
    - Intentar ser lo más **sistemático** posible en identificar un **conjunto representativo de comportamientos** del programa
      - Determinados por subclases del dominio de entradas, escenarios, estados,…
  - **Objetivo**
    - "romper" el programa, encontrar el mayor número de fallos posible
  - De forma general, existen **dos enfoques** diferentes:
    - **Caja Negra** (**Funcional**)

      Los casos de prueba se basan sólo en el comportamiento de entrada/salida
    - **Caja Blanca** (**Estructural**)

      Basadas en información sobre cómo el software ha sido diseñado o codificado



- **Caja Negra**
  - Orientadas a los requisitos funcionales



¿$y_i=f(x_i)$, para todo *i*?

Buscan asegurar que:

- Se ha ingresado toda clase de entrada
- Que la salida observada = esperada

3.Pruebas – Técnicas

- **Particionamiento Equivalente**
  - El dominio de las **entradas** se divide en subconjuntos equivalentes respecto de una relación especificada (**clases de equivalencia**):
    - La prueba de un valor representativo de una clase permite suponer «razonablemente» que el resultado obtenido será el mismo que para otro valor de la clase
  - Se realiza un conjunto representativo de casos de prueba para cada clase de equivalencia

- **Particionamiento Equivalente**
  - **Ejemplo**: Entrada de un Programa
    - Código de Área: Número de tres cifras que no comienza ni por 0 ni por 1
    - Nombre: Seis caracteres
    - Orden: cheque, depósito, pago factura, retirada de fondos

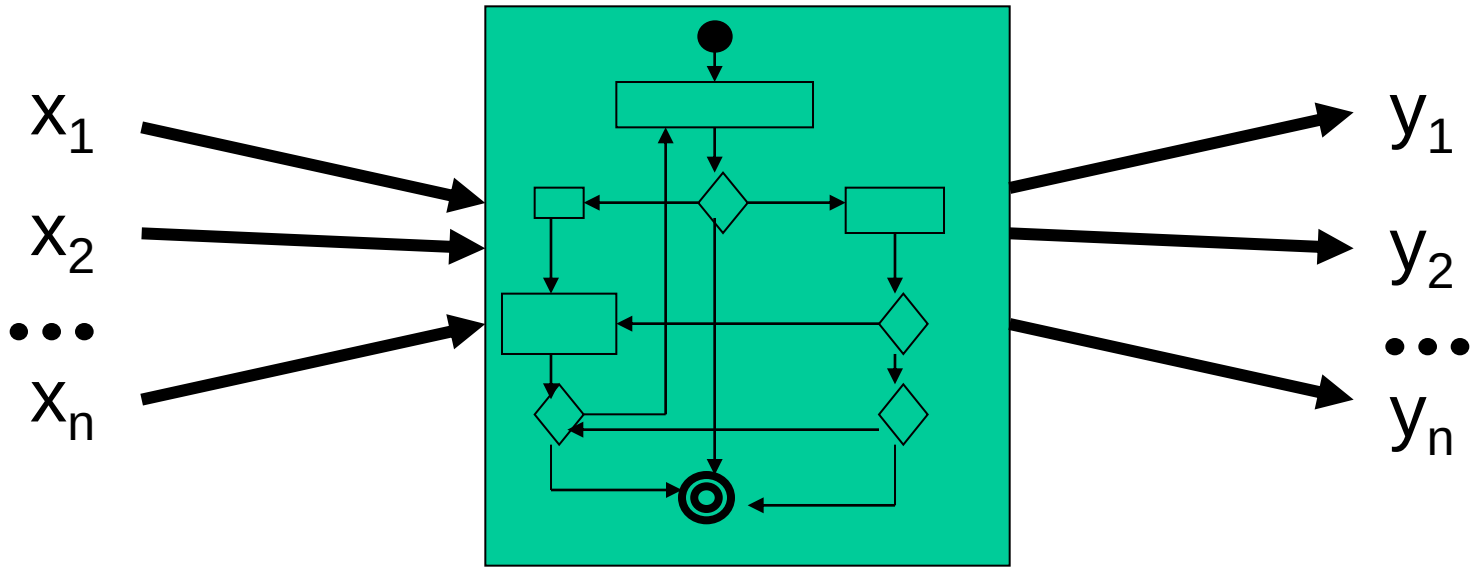
Condición de entrada	Clases válidas	Clases inválidas
Código área		
Nombre para identificar la operación		
Orden		

3.Pruebas – Técnicas

- **Análisis de Valores Límite**
  - Similar a la anterior pero los valores de entrada de los casos de prueba se eligen en las cercanías de los **límites** de los dominios de **entrada** de las variables
  - **Suposición**: Muchos defectos tienden a concentrarse cerca de los valores extremos de las entradas
  - Extensión: **Pruebas de Robustez**, con casos fuera de los dominios de entrada

  - Si aplicamos AVL en el ejemplo anterior ¿qué valores para los casos de prueba seleccionaríamos?

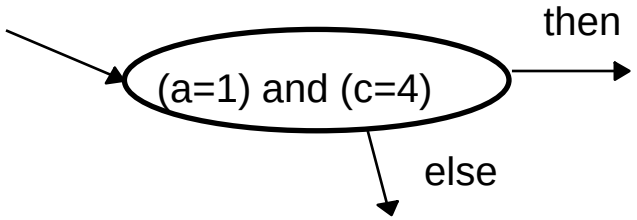
- **Caja Blanca**
  - De alguna manera, me interesa conocer la **cobertura** alcanzada por mis casos de prueba dentro de la unidad de prueba

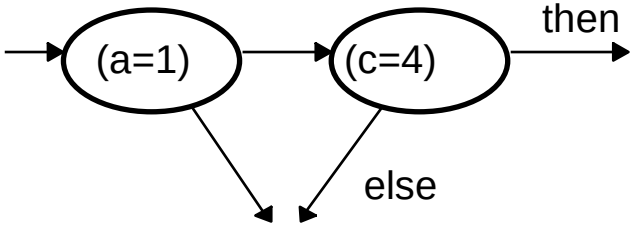


3.Pruebas – Técnicas

- Criterios de Cobertura:
  - ☑ **Cobertura de sentencias**. Que cada sentencia se ejecute al menos una vez.
  - ☑ **Cobertura de decisiones**. Que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.
  - ☑ **Cobertura de condiciones**. Que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez.
  - ☑ **Criterio de decisión/condición**. Que se cumplan a la vez el criterio de condiciones y el de decisiones.
  - ☑ **Criterio de condición múltiple**. La evaluación de las condiciones de cada decisión no se realiza de forma simultánea.

Descomposición de una Decisión Multicondicional

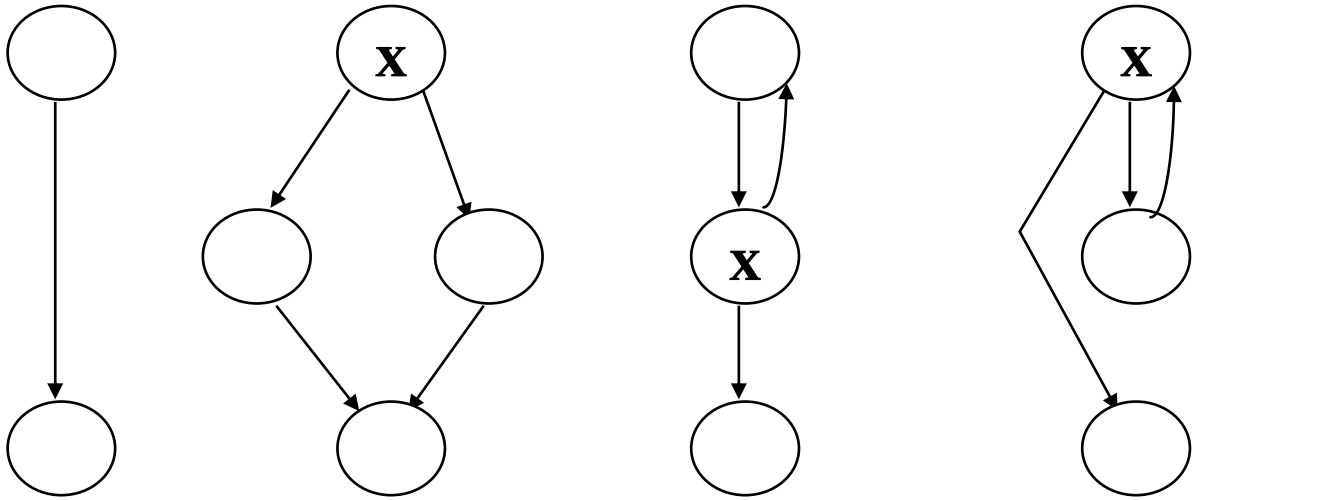




3.Pruebas – Técnicas

- **Grafos de Flujo**
  - Utilizados para la representación del flujo de control
  - La estructura de control sirve de base para obtener los casos de prueba

*El diseño de casos de prueba tiene que estar basado en la elección de **caminos** importantes que ofrezcan una seguridad aceptable de que se descubren defectos*

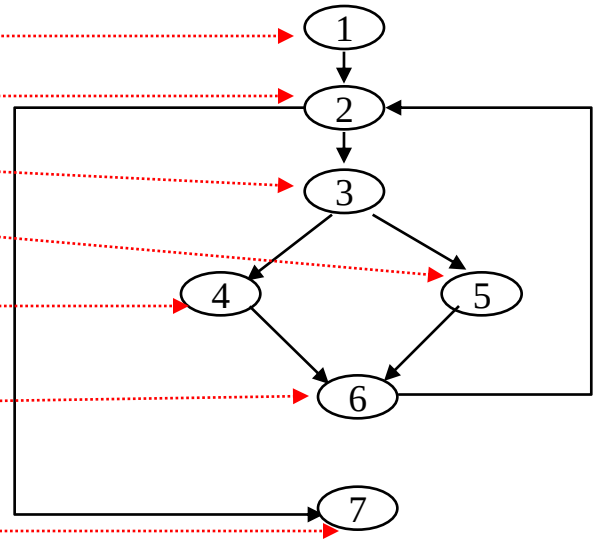


Secuencia | **Si x entonces... (If x then...else...)** | **Hacer... hasta x (Do...until x)** | **Mientras x hacer... (While x do...)**

3.Pruebas – Técnicas

- Grafo de Flujo de un Programa (Pseudocódigo)

```
Abrir archivo;
WHILE (haya registros) DO
        IF (cliente) THEN
                Mostrar cliente;
        ELSE
                Mostrar registro;
        ENDIF;
Incrementar registro;
 ENDWHILE;
Cerrar archivos.
```



Complejidad ciclomática:

- Indicador del número de caminos independientes de un grafo
- Límite mínimo de número de casos de prueba para un programa

Cálculo de complejidad ciclomática

- V(G) = arcos – nodos + 2
- V(G) = número de regiones cerradas
- V(G) = número de nodos condición + 1

Posible conjunto de caminos

- 1 – 2 – 7
- 1 – 2 – 3 – 4 – 6
- 1 – 2 – 3 – 5 – 6

3.Pruebas – Técnicas

- Grafo de Flujo de un Programa (Pseudocódigo)

```
Abrir archivos;
Leer archivo ventas, al final indicar no más registros;
Limpiar línea de impresión;
WHILE  (haya registros ventas) DO
  Total nacional = 0;
  Total extranjero = 0;
    WHILE   (haya reg. ventas)   y      (mismo producto)
      IF   (nacional) THEN
        Sumar venta nacional a total nacional
      ELSE
        Sumar venta extranjero a total extranjero
      ENDIF;
      Leer archivo ventas, al final indicar no más registros;
    ENDWHILE;
    Escribir línea de listado;
    Limpiar área de impresión;
ENDWHILE;
Cerrar archivos.
```

3.Pruebas – Técnicas

- Otro criterio más preciso clasifica las técnicas de prueba según la fuente a partir de la que son generados los casos de prueba:

- **Experiencia** del ingeniero de pruebas
  - Ad-hoc
  - Exploratorias
- **Especificaciones**
  - Particionamiento equivalente
  - Análisis de valores límite
  - Tablas de decisión
  - Máquinas de estados finitos
  - Especificación formal
  - Aleatorias
- **Estructura del código**
  - Flujo de control
  - Flujo de datos
- **Defectos** que se quieren descubrir
  - Conjetura de errores
  - Mutación
- El **campo de uso**
  - Perfil operacional
  - SRET (guiadas por objetivos de fiabilidad)
- La **naturaleza de la aplicación**
  - Orientado a Objetos
  - Basado en Componentes
  - Basado en Web
  - Interfaz de Usuario
  - Programas Concurrentes
  - Conformidad de Protocolos
  - Sistemas en Tiempo Real
  - Sistemas Críticos (seguridad)

3.Pruebas – Técnicas

- **Basadas en la Experiencia**
  - **Ad Hoc**
    - Las pruebas dependen totalmente de la habilidad, intuición y experiencia con programas similares del **ingeniero de pruebas**
  - **Exploratorias**
    - A la vez, se lleva a cabo aprendizaje, diseño de pruebas y ejecución de pruebas
    - Las pruebas se diseñan, ejecutan y modifican de forma dinámica, **sobre la marcha**
    - La eficiencia depende fundamentalmente del conocimiento del ingeniero de pruebas

3.Pruebas – Técnicas

- **Basadas en la Especificación**
  - **Tablas de Decisión**
    - Se usan tablas de decisión para representar relaciones lógicas entre condiciones (entradas) y acciones (salidas)
    - Los casos de prueba se derivan sistemáticamente de cada **combinación de condiciones y acciones**
  - **Máquinas de Estados Finitos**
    - Los programas se modelan como máquinas de estados finitos
    - Los casos de prueba pueden ser seleccionados de forma que cubren los **estados y las transiciones** entre ellos

3.Pruebas – Técnicas

- **Basadas en la Especificación**
  - **Especificación Formal**
    - Disponer de las especificaciones en un **lenguaje formal** permite la derivación automática de casos de prueba
    - A la vez, se provee una salida de referencia (oráculo) para chequear los resultados de las pruebas
      - Basadas en Modelos
      - Basadas en Especificaciones Algebraicas
  - **Aleatorias**
    - Las Pruebas son generadas de forma plenamente **aleatoria**
    - Requiere el conocimiento del dominio de las entradas
      - Generación aleatoria de datos de entrada con la secuencia y la frecuencia con las que podrían aparecer en la práctica

3.Pruebas – Técnicas

- **Basadas en el Código**
  - **Flujo de Control**
    - Se usan criterios que buscan cubrir todas las sentencias o bloques de sentencias de un programa
    - El criterio más fuerte es la prueba de caminos (*testing path*)
      - Ejecución de todos los caminos de flujo de control entrada-salida
    - Otros criterios menos exigentes:
      - Prueba de sentencias
      - Prueba de ramas (branch)
      - Prueba de condiciones/decisiones
    - Los resultados de estas pruebas se establecen en **porcentaje de cobertura**

3.Pruebas – Técnicas

- **Basadas en el Código**
  - **Flujo de Datos**
    - El diagrama de flujo de control es anotado con información sobre cómo se definen, usan y eliminan las variables del programa
    - El criterio más fuerte busca probar todos los caminos de definición-uso:
      - Para cada variable, se debe ejecutar cada segmento de camino de flujo de control desde la definición de la variable a su uso
    - Criterios menos exigentes:
      - Prueba de todas las definiciones
      - Prueba de todos los usos

- **Basadas en Defectos**
  - "inventan" los casos de prueba con el objetivo de descubrir categorías de defectos probables o previstos
  - **Conjetura de Errores**
  - Los casos de prueba se diseñan intentando **descubrir** los defectos más plausibles del programa
    1. Enumerar una **lista** de posibles equivocaciones que pueden cometer los desarrolladores y de las situaciones propensas a ciertos errores
       - Ejemplo: El valor cero es una situación propensa a error
    2. Generar los casos de prueba en base a dicha lista (se suelen corresponder con defectos que aparecen comúnmente y no con aspectos funcionales)
  - Fuente: Historia de defectos en proyectos previos

3.Pruebas – Técnicas

- **Basadas en Defectos**
  - **Mutación**
    - Un mutante es una **versión ligeramente modificada** de un programa. La diferencia es un pequeño **cambio sintáctico**
    - Cada caso de prueba se aplica al original y a los mutantes generados
    - **Asunción**: Mirando defectos sintácticos sencillos se encuentran defectos reales más complejos
    - Para ser eficiente se requieren muchos mutantes, que deben ser **generados automáticamente** de forma sistemática
      - Hay herramientas disponibles para ello

3.Pruebas – Técnicas

- **Basadas en el Uso**
  - **Perfil Operacional**
    - Se reproduce y se prueba el entorno operativo en que deberá funcionar el programa
    - Se trata de inferir, a partir de los resultados observados, la futura fiabilidad del software cuando esté en uso
  - **SRET** (*Software Reliability Engineered Testing*)
    - Método en el cual las pruebas son "diseñadas y guiadas por objetivos de fiabilidad y criticidad de las diferentes funcionalidades"