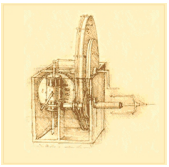
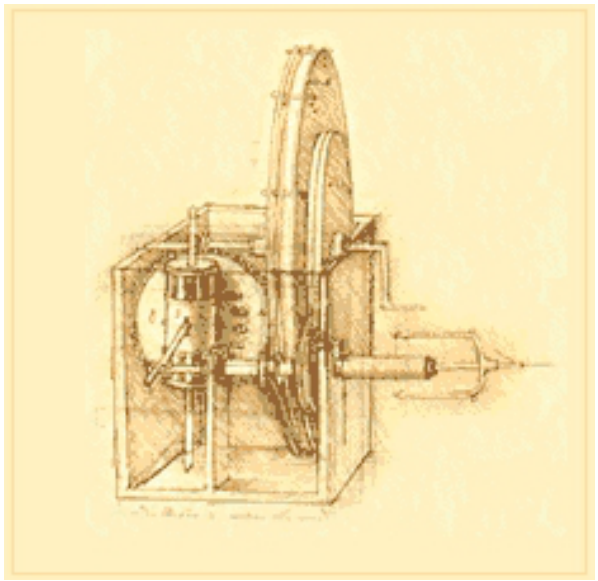
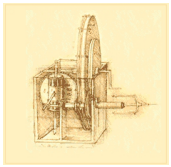


Tema 4.2 - Validación y verificación de software (V&V)



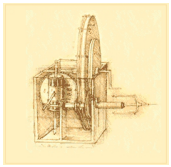
Bibliografía: **[PRES13 capítulo 16, 17 y 18]**
[SOMM05 (7ª edición) capítulos 22 y 23]



Contenido

Tema 4.2 - Validación y verificación de software (V&V)

- Concepto de V&V
- Planteamiento
- Tareas de la V&V
- Inspecciones del software
- Prueba del software
  - Conceptos y niveles
  - Pruebas de Aceptación/Validación
  - Prueba de defectos
    - Prueba del sistema
    - Prueba de Integración
    - Prueba de Unidad
    - Técnicas de prueba de Unidad

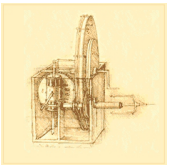


Concepto de V&V

Conjunto de procesos de comprobación y análisis que aseguran que el software que se desarrolla está acorde a su especificación y cumple las necesidades de los clientes

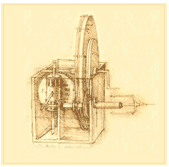
Verificación → ¿Estamos construyendo el producto correctamente?
Coherencia interna y conforme con su especificación.

Validación → *¿Estamos construyendo el producto correcto?*
Producto que cumple con las expectativas del cliente/usuario.



Planteamiento

- Se trata de **encontrar los defectos** de los productos software, no de corroborar que, *como cabía esperar,* todo está bien.
- Debemos asumir que nuestro programas tienen errores, y buscarlos, a sabiendas que **no los vamos a encontrar todos**, la perspectiva no es muy atractiva.
- Es necesario acometer con una **mentalidad positiva** esta **tarea destructiva**. Una actividad de verificación y validación alcanza el éxito cuando permite encontrar errores.
- Es necesario distanciarse conceptualmente respecto a la perspectiva desde la que se abordó el desarrollo, por ello es conveniente que la realice un **equipo distinto al de desarrollo**.
- Consume entre el 30% y 40% del esfuerzo de desarrollo.



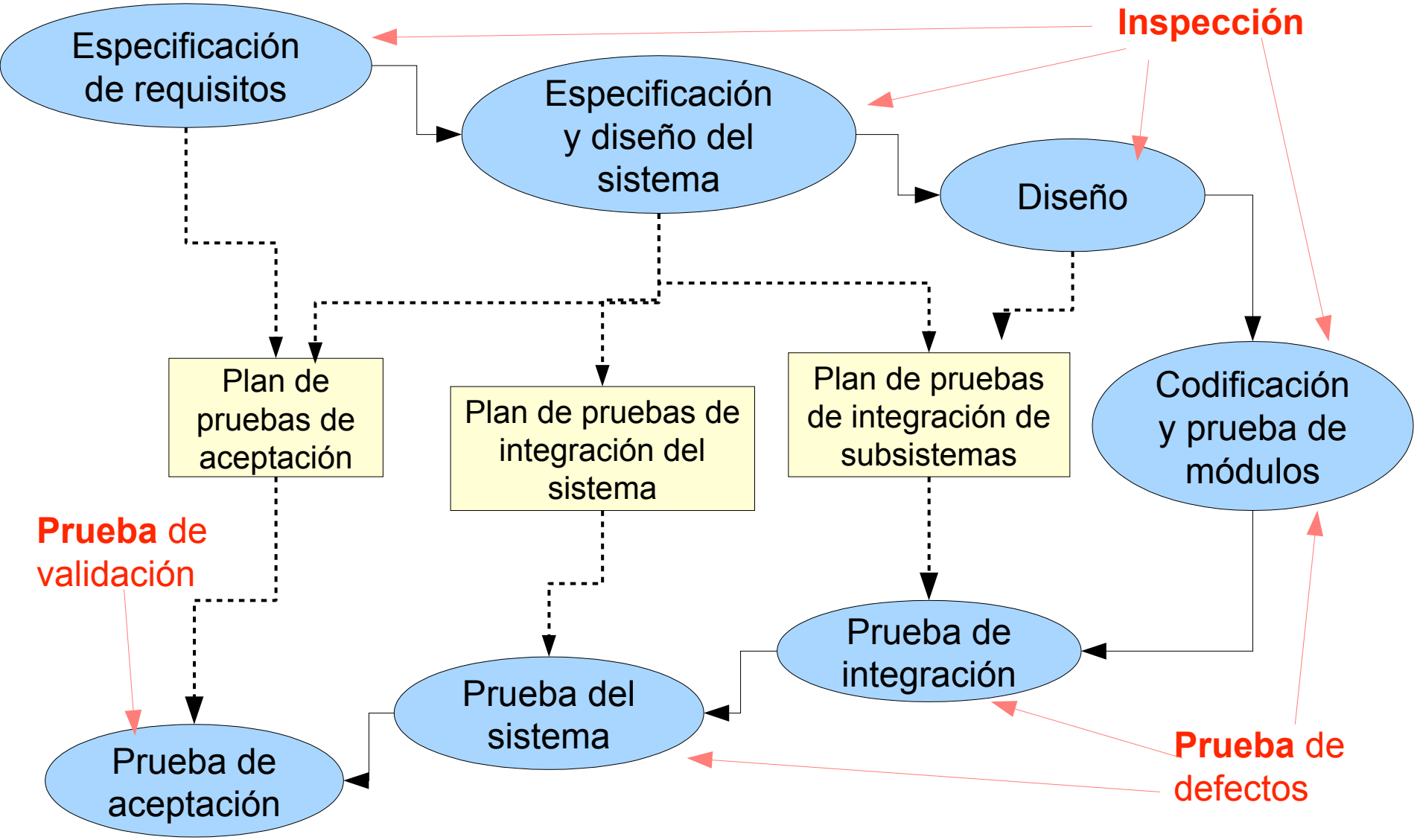
Tareas de V&V

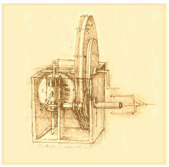
Son complementarias y tienen lugar en cada una de las etapas del proceso de producción del software. Éstas en general son:

- **Inspección del software**: Analiza y comprueba las representaciones del sistema: documentos, diagramas, código... Son técnicas estáticas, no necesitan de la ejecución del sistema.
- **Prueba del software**: Ejecución de la implementación con datos de prueba, para comprobar que funciona cómo se esperaba que lo hiciese, son técnicas dinámicas. Pueden ser:
  - **Pruebas de validación**: intentan demostrar que el software es el que el cliente quiere.
  - **Pruebas de defectos:** hallan inconsistencias entre un sistema software y su especificación.

Nota: *No confundir la V&V con la depuración de errores*

Tareas de la V&V

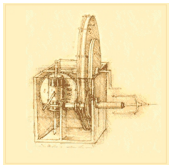




Inspecciones del software

Consisten en **revisiones sistemáticas** de los documentos, modelos, código fuente... con el único objetivo de **detectar fallos**.

- Permiten detectar entre un **60% y un 90% de los fallos a** unos costes mucho más bajos que las pruebas dinámicas.
- Permiten detectar **múltiples defectos** en una simple inspección, mientras que las pruebas solo suelen detectar un tipo fallo por prueba.
- Las inspecciones son **útiles** para detectar los **fallos de módulos**, pero no detectan fallos a nivel de sistema, que ha de hacerse con pruebas.
- La inspecciones **no son útiles** para la detección de niveles de **fiabilidad** y evaluación de **fallos no funcionales**.



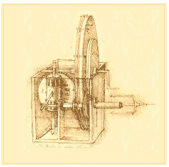
Inspecciones del software

Inspección de código: Analizadores estáticos

Son herramientas de software que **rastrean el texto fuente** de un programa, en busca de errores no detectados por el compilador (la mayoría de los compiladores ya lo incluye).

Aspectos analizados:

- **Flujo de control**: Identifica y señala bucles con múltiples puntos de salida y secciones de código no alcanzable.
- **Utilización de datos**: Señala el uso de las variables del programa: Variables sin utilización previa, que se declaran dos veces, declaradas y nunca utilizadas. Condiciones lógicas con valor invariante, etc.
- **Análisis de interfaces**: Verifica la declaración de las operaciones y su invocación. Esto es inútil en lenguajes con tipo estático (Java).
- **Caminos de ejecución**: Identifica todas las posibles caminos de ejecución del programa y presenta las sentencias ejecutadas en cada camino.

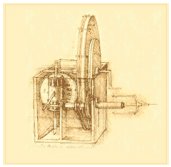


Prueba: Concepto y niveles

Concepto: Ejecución de la implementación con una serie de datos de prueba y examen de las respuestas y comportamiento funcional, para comprobar que se lleva a cabo conforme a lo especificado.

Niveles:

- **Prueba de validación** (o de aceptación): Prueba del sistema en el entorno real de trabajo con intervención del usuario final.
- **Prueba de defectos:**
  - **Prueba de sistema**: Prueba el sistema como un todo.
  - **Prueba de integración o de subsistemas**: Prueba agrupaciones de módulos relacionados (funciones o clases).
  - **Prueba de unidades**: Prueba de cada módulo (función, método o clase) de forma independiente.

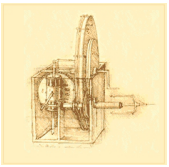


Prueba de Aceptación/Validación

Proceso de la prueba de aceptación

Figura de SOMM05 (7ª edicion): http://www.softwareengineering-9.com/

- Basadas en los criterios de aceptación
- Resultado de la prueba: Aceptación o rechazo del sistema
- Pueden ser:
  - Pruebas alfa (entorno de desarrollo)
  - Pruebas beta (entorno del cliente)



Prueba de defectos

- La prueba de defectos va **dirigida a** los niveles de unidad, integración y sistema.
- El **objetivo** de las pruebas de defecto es detectar los defectos latentes de un sistema software antes de entregar el producto.
- Una prueba de defectos **exitosa** es aquella que **descubre fallos**, esto es, un comportamiento contrario a la especificación.
- La pruebas de defectos **demuestran la existencia** de un fallo, y **no la ausencia** de cualquier fallo.
- Las pruebas **exhaustivas no son posibles** y deben sustituirse por subconjuntos de casos de prueba.

Prueba de defectos: Proceso

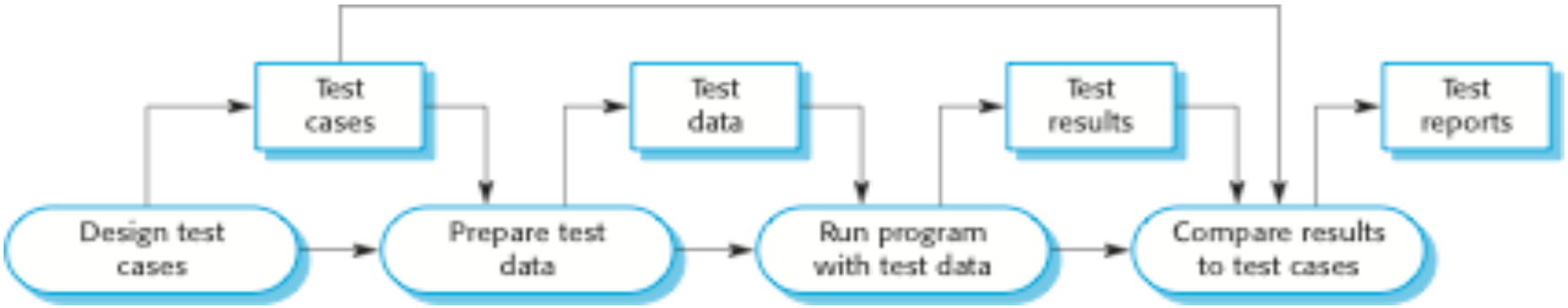
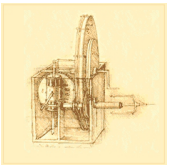


Figura de SOMM11 (7ª edicion): http://www.softwareengineering-9.com/

- Los **casos de prueba** son especificaciones de las entradas a la prueba y de la salida esperada del sistema, más una declaración de lo que se prueba.
- Los **datos de prueba** son las entradas seleccionadas, que cumplen con lo especificado en los casos de prueba.

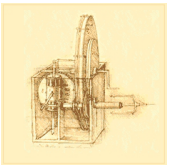


Prueba de defectos: **Prueba del sistema**

Los casos de prueba a nivel de sistema van **dirigidos a** detectar problemas con:

- Recuperación
- Seguridad
- Resistencia
- Rendimiento
- …

En general con la detección de fallos o no concordancia con los **requisitos no funcionales**.



Prueba de defectos: **Prueba de integración**

Se prueba la respuesta de **grupos de módulos** interconectados a fin de detectar fallos resultantes de su interacción, junto con acceso incoherente a estructuras de datos globales, tiempos de respuesta...

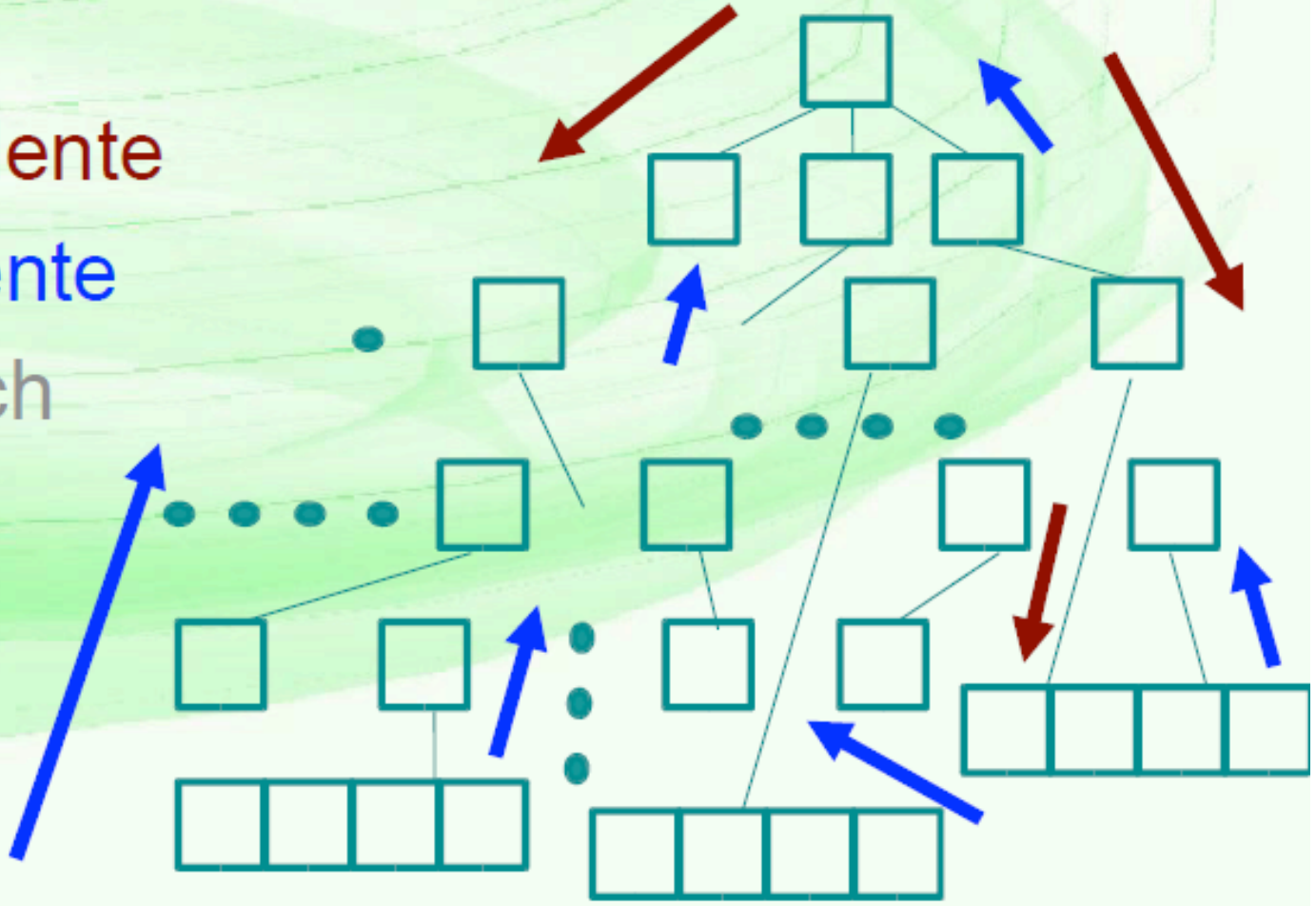
La principal dificultad de las pruebas de integración es la localización de los fallos, para facilitarla se utilizan **estrategias** incrementales. Estas pueden ser:

- Ascendente
- Descendente
- Mixta/Sandwich

Prueba de defectos: **Prueba de integración**

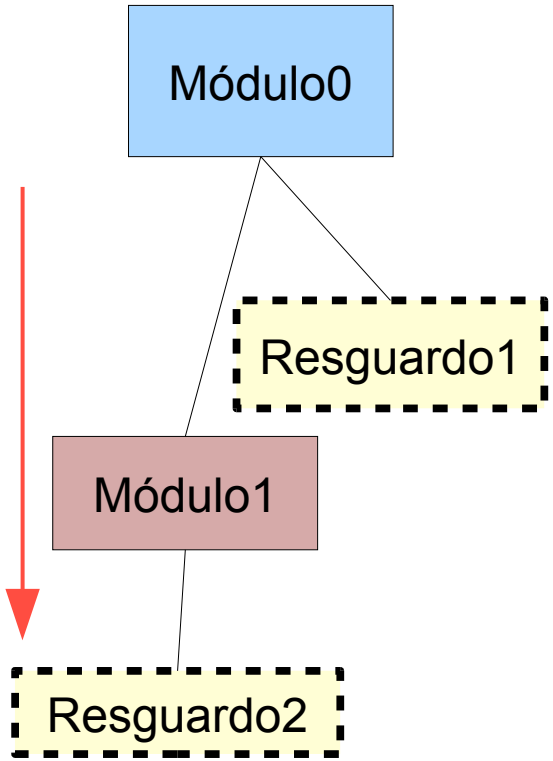
Estrategias:

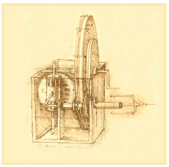
- descendente
- ascendente
- sandwitch



Estrategia descendente: Procedimiento

1. Recorrer la estructura de arriba hacia abajo, avanzando en profundidad o en anchura
2. Para probar un módulo: tomar el módulo del que depende como driver, sustituir los módulos dependientes por resguardos y realizar las pruebas específicas del módulo
3. Progresar substituyendo resguardos por módulos reales realizando pruebas específicas para cada nuevo módulo y repitiendo las realizadas previamente (pruebas regresivas)

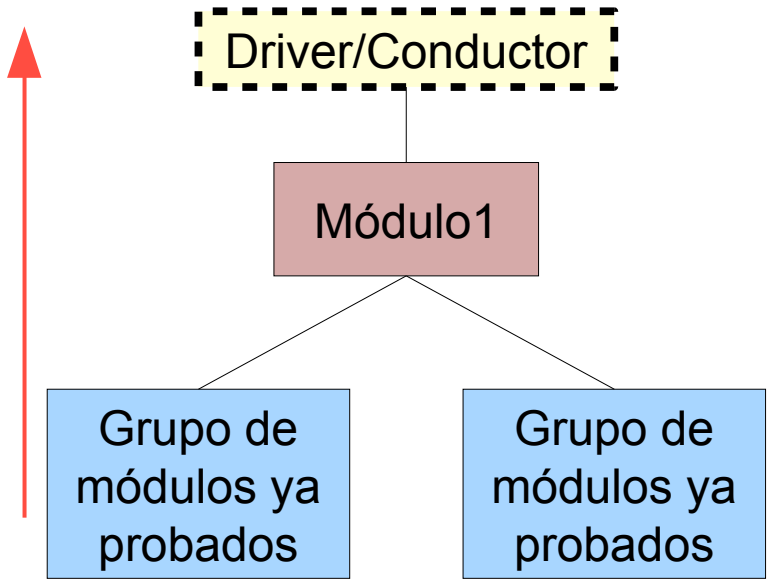




Prueba de defectos: **Prueba de integración**

Estrategia ascendente: Procedimiento

1. Recorrer la estructura de abajo hacia arriba
2. Agrupar los módulos inferiores
3. Preparar un driver para cada grupo y realizar sus pruebas
4. Progresar sustituyendo los driver por módulos reales realizando pruebas específicas y regresivas



Ventaja:

- No es necesario elaborar resguardos

Inconveniente:

- Los módulos más importantes se prueban al final; esto genera incertidumbre

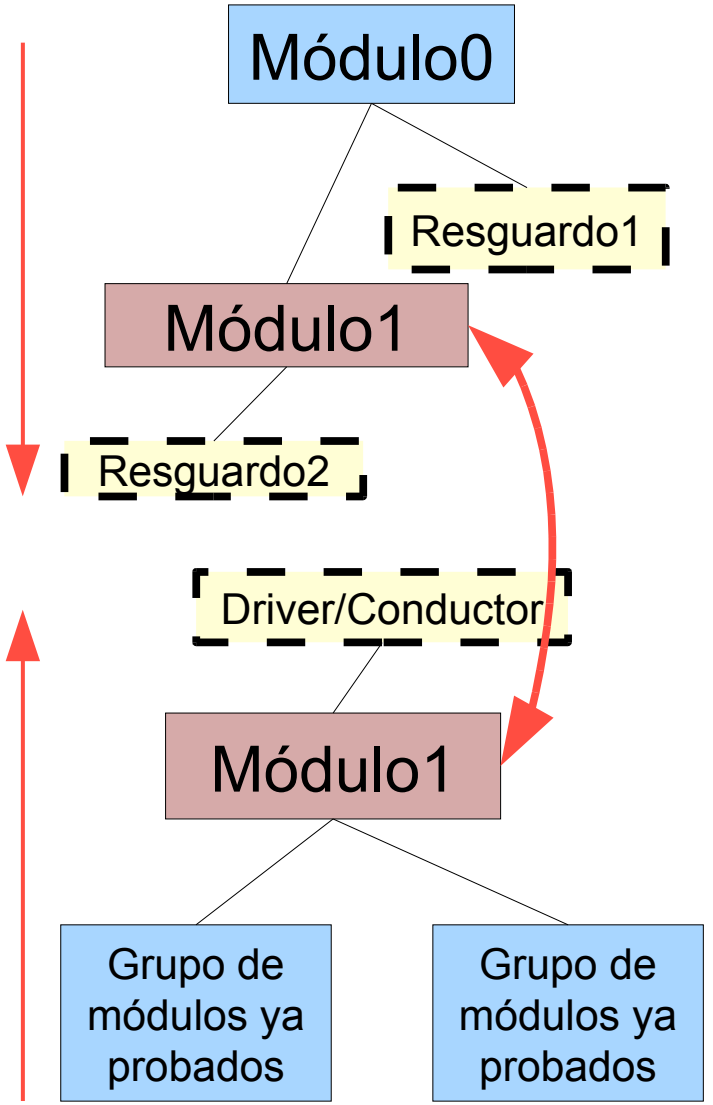
Estrategia mixta

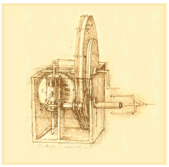
Combinamos:

- La estrategia descendente para los módulos superiores
- La estrategia ascendente para los módulos inferiores

Con esta estrategia **conseguimos**:

- Las ventajas de las dos anteriores
- El número de resguardos y conductores que tenemos que desarrollar en menor





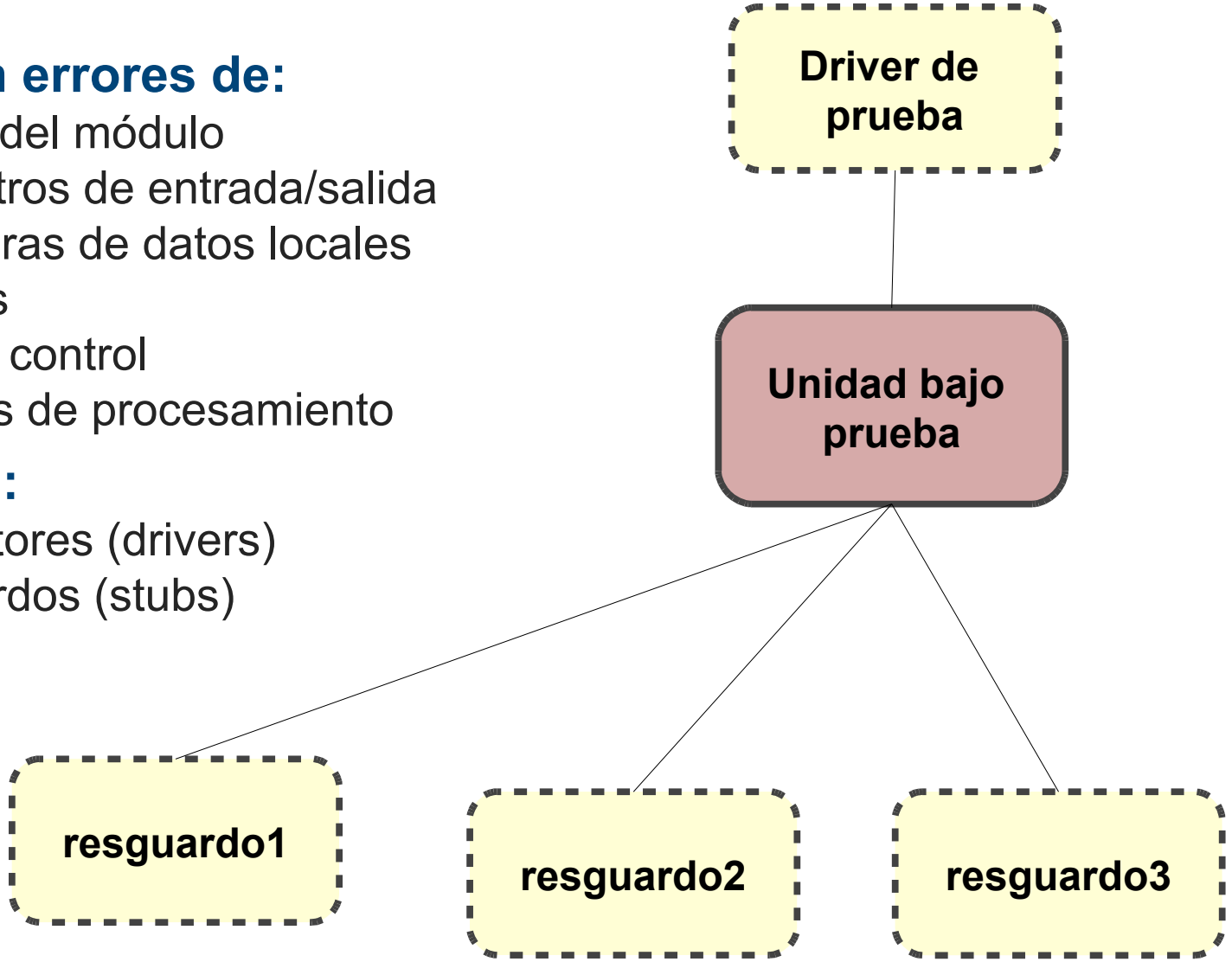
Prueba de defectos: **Prueba de unidad**

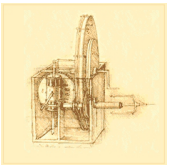
Se buscan errores de:

- Interfaz del módulo
- Parámetros de entrada/salida
- Estructuras de datos locales
- Cálculos
- Flujo de control
- Caminos de procesamiento

Requieren:

- Conductores (drivers)
- Resguardos (stubs)





Prueba de defectos: **Técnicas de Prueba de Unidad**

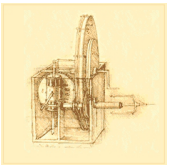
Las **técnicas de prueba** ayudan a definir conjuntos de casos de prueba, aplicando cierto criterio

Los Caso de prueba especifican la prueba en términos de:

- Los valores de entrada a suministrar
- Los valores de salida correctos

Tipos de técnicas de prueba:

- **Caja negra**: los casos se deducen de las interfaces y especificaciones del módulo
- **Caja blanca**: los casos se deducen del contenido o interior del módulo



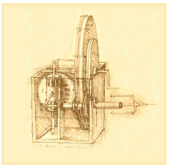
Técnicas de caja negra

Las técnicas de caja negra **permiten** detectar:

- funcionamiento incorrecto o incompleto
- errores en la interfaz
- errores accesos estructuras de datos externas
- problemas de rendimiento
- errores de inicio y terminación

Criterios de selección de casos de prueba:

- valores representativos de conjuntos de datos fronteras
- valores o combinaciones de valores conflictivos
- capacidad de proceso



Técnicas de caja negra: Particiones Equivalentes

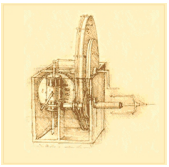
Objetivo:

Ejecutar los módulos con todos los posibles valores distintos de los argumentos de entrada al módulo

Pasos a seguir:

1. Encontrar todas particiones equivalentes (clases de equivalencia) de todos los valores de entrada, es decir la especificación de los **casos de prueba**

2. Derivar los **datos de prueba** eligiendo al menos un valor de cada clase de equivalencia

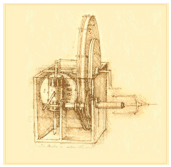


Técnicas de caja negra: Particiones Equivalentes

Reglas para establecer las clases de equivalencia:

1. Si una entrada está limitada a un **rango de valores**, hay como mínimo **tres clases de equivalencia**, una con valores menores que el rango, otra con valores dentro del rango y otra con valores mayores que el rango.

2. Si una entrada válida está dentro de un **conjunto de valores discreto**, hay **dos clases de equivalencia**, una conteniendo los valores válidos y otra conteniendo cualquier otro valor de entrada.



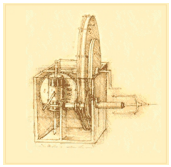
Técnicas de caja negra: Particiones Equivalentes

Ejemplo

Si tenemos un módulo con los siguientes argumentos:

NombreArticulo: String entre 2 y 15 caracteres alfanuméricos

Peso [5]: Array de 5 elementos reales, donde cada uno de los elementos representa el peso, entre 0 y 10.000 gr., que se puede tener del artículo en cuestión. Estos pesos están ordenados de menor a mayor y si el artículo tiene sólo tres pesos los dos primeros elementos estarán a 0 y los tres últimos a valores distinto de cero



Técnicas de caja negra: Particiones Equivalentes

Ejemplo: Clases de equivalencia o casos de prueba

***NombreArticulo**:*

1. Alfanumérico (válido): AcdEf4
2. No alfanumérico (no válido): A$%!1

Longitud NombreArticulo:

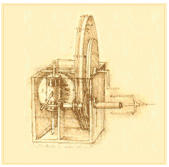
3. 2<=L<=15 (válido): afdHteKJN14
4. L<2 (no válido): a
5. L>15 (no válido):aaaaaaaaaaaaaaaaaaaaaaaaa

Rango de valores para el peso:

6. Peso < 0 gr. (invalida): -2
7. 0<= Peso <=10.000 gr. (válida): 500
8. Peso > 10.000 (invalida): 11.000

Orden:

9. Elementos ordenados (válida): [0,0,1,5,10]
10. Elementos no ordenados (inválida): [1,0,10,5,0]



Técnicas de caja negra: Particiones Equivalentes

Ejemplo: Datos de Prueba

Caso 1: (“abcd”,[0,1,2,3,4])

Resultado esperado: Ejecución del módulo sin problemas.

Caso 2: (“abcd”,[0,1000,2000,3000,11000])

Resultado esperado: Salida por error (“peso no válido”)

Caso 3: (“abcd”, [-1,0,2,3,4])

Resultado esperado: Salida por error (“peso no válido”)

Caso 4: (“abcd”, [1,0,10,5,0)])

Resultado esperado: Salida por error (“valores desordenados”)

Caso 5: (“$%&”,[0,0,0,0,1])

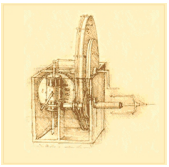
Resultado esperado: Salida por error (“nombre no válido”)

Caso 6: (“a”,[0,0,0,0,1])

Resultado esperado: Salida por error (“long. nombre no valida”)

Caso 7: (“aaaaaaaaaaaaaaaaa”,[0,0,0,0,1])

Resultado esperado: Salida por error (“long. nombre no valida”)



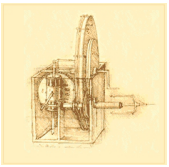
Técnicas de caja blanca

Las técnicas de caja blanca **permiten** detectar errores en:

- Estructuras de datos locales
- Cálculos
- Flujo de control
- Caminos de procesamiento

Criterios de selección de casos de prueba:

- Caminos independientes
- Valores de las condiciones
- Bucles dentro y fuera de sus límites operacionales
- Estructuras de datos



Técnica de caja blanca: El camino básico

Objetivo:

Ejecutar todas las líneas de código de un módulo al menos una vez

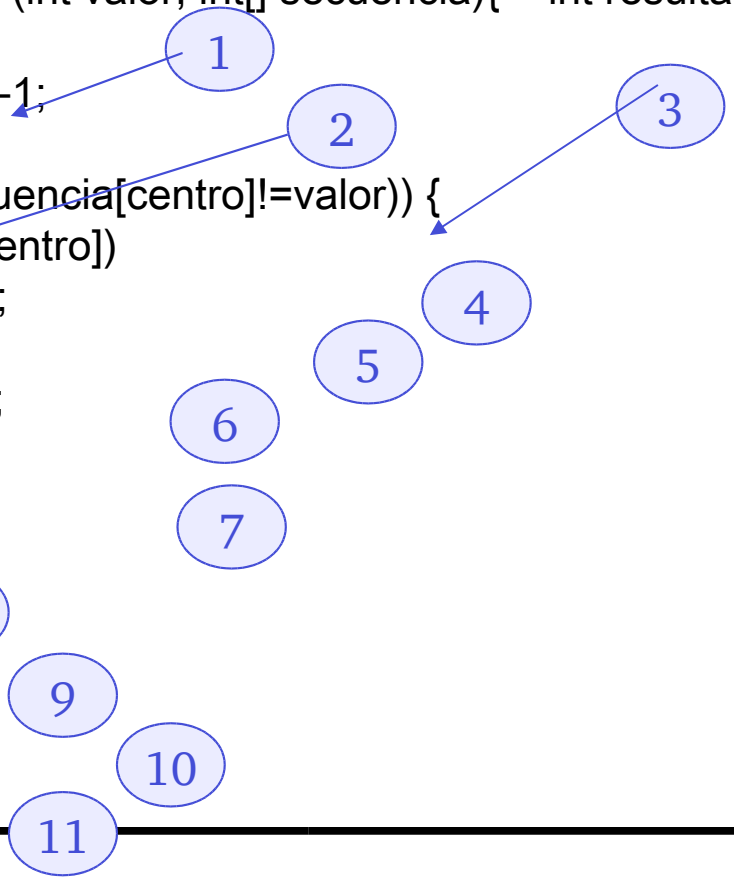
Pasos a seguir*:*

1. Representar el flujo de control del módulo mediante un Grafo de Flujo.
2. Determinar la complejidad del Grafo de Flujo (complejidad ciclomática de "MC Cabe").
3. Usar esa medida como guía para definir el conjunto de caminos básicos de ejecución (Especificación de los casos de prueba)
4. Derivar los datos de prueba a partir del conjunto de caminos básico de ejecución.
5. Verificar que los casos de prueba diseñados no dejan ninguna arista (línea de código) sin pasar.

Técnica de caja blanca: El camino básico

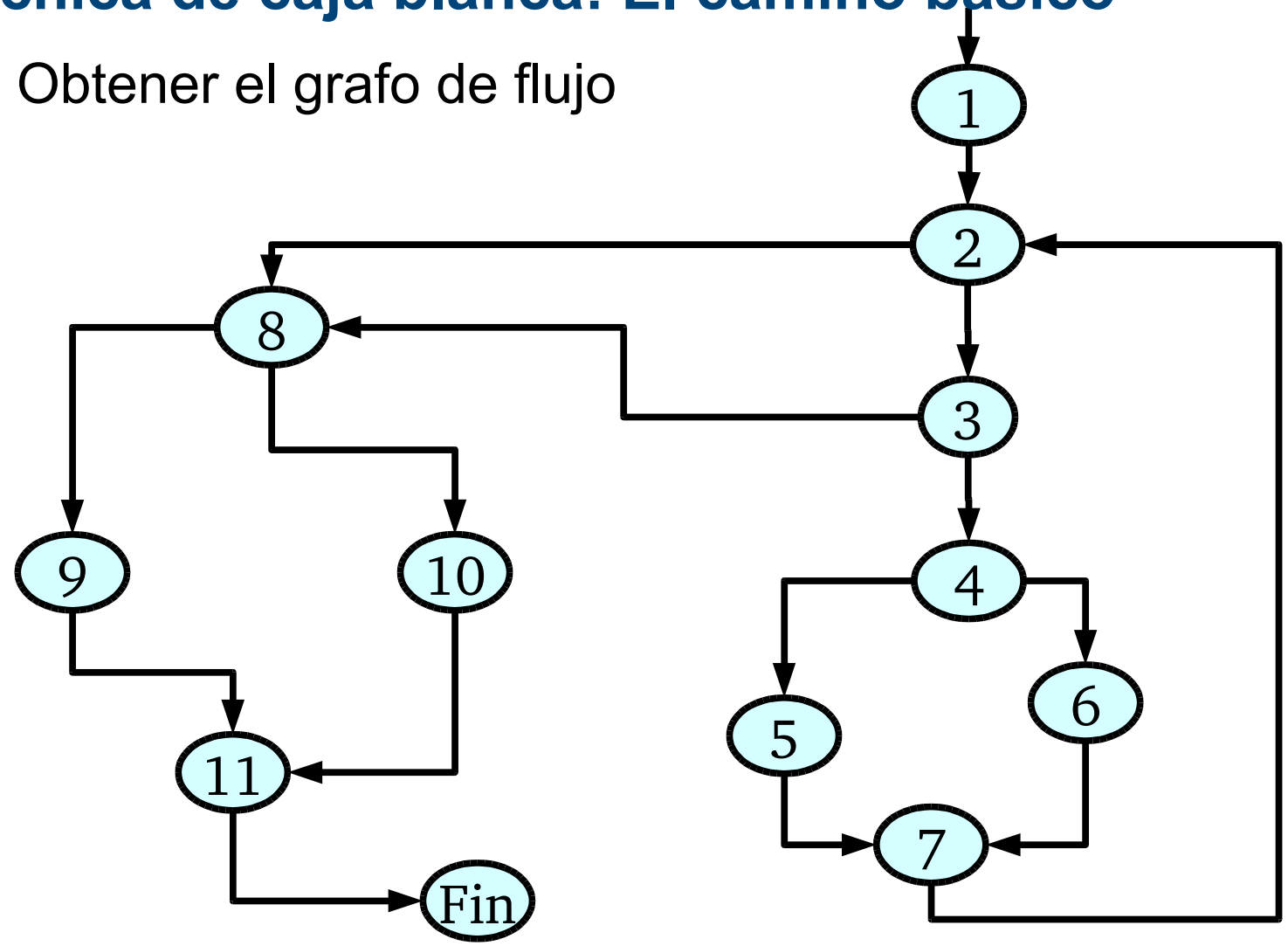
1. Obtener el grafo de flujo

```
public static int binarySearch (int valor, int[] secuencia){    int resultado;
    int izq = 0;
    int der = secuencia.length-1;
    int centro = (izq+der)/2;
    while ((izq<=der) && (secuencia[centro]!=valor)) {
        if (valor<secuencia[centro])
            der = centro – 1;
        else
            izq = centro + 1;
        centro = (izq+der)/2;
    }
    if (izq>der)
        resultado = -1;
    else
        resultado = centro;
    return resultado;
}
```



Técnica de caja blanca: El camino básico

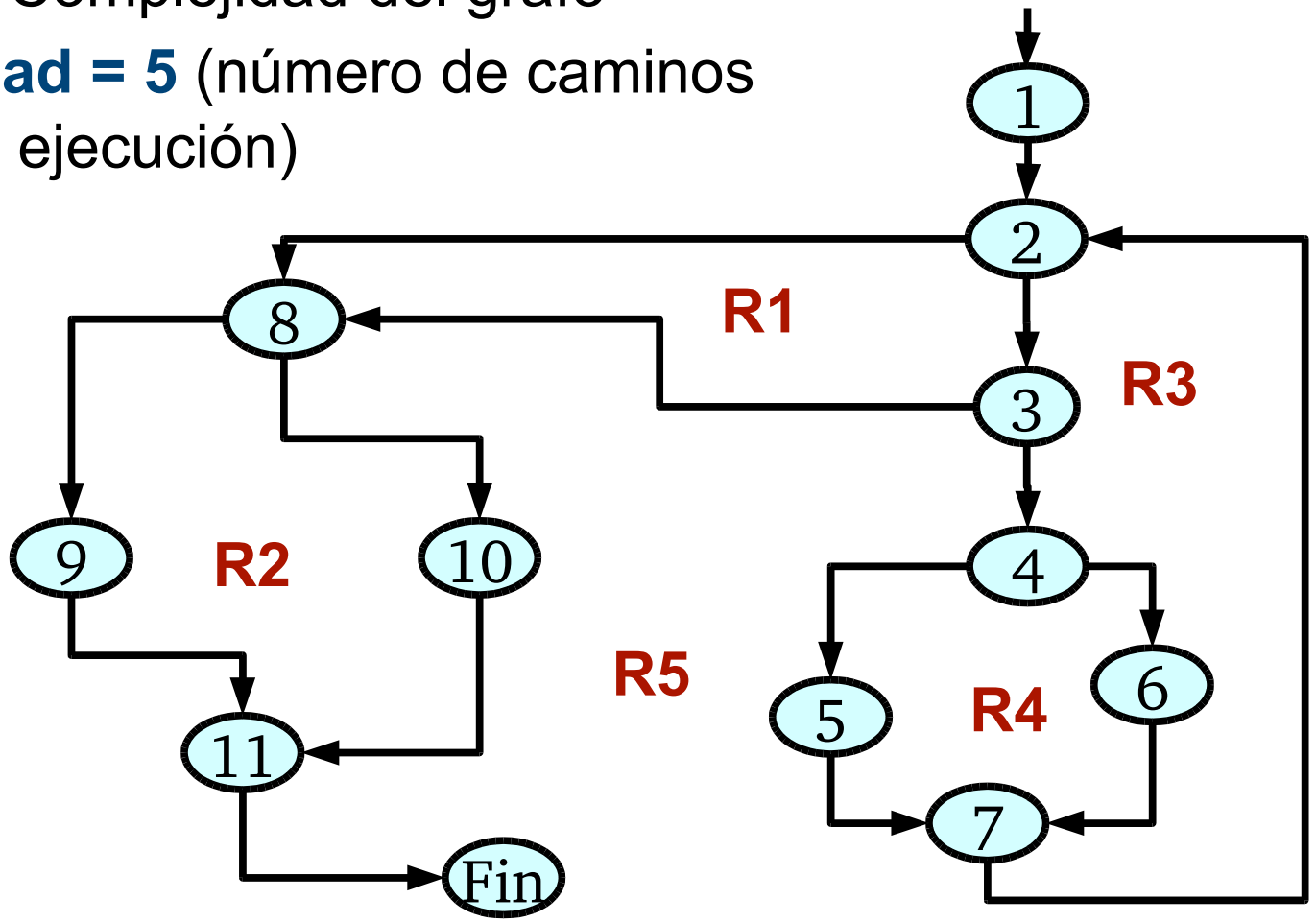
1. Obtener el grafo de flujo



Técnica de caja blanca: El camino básico

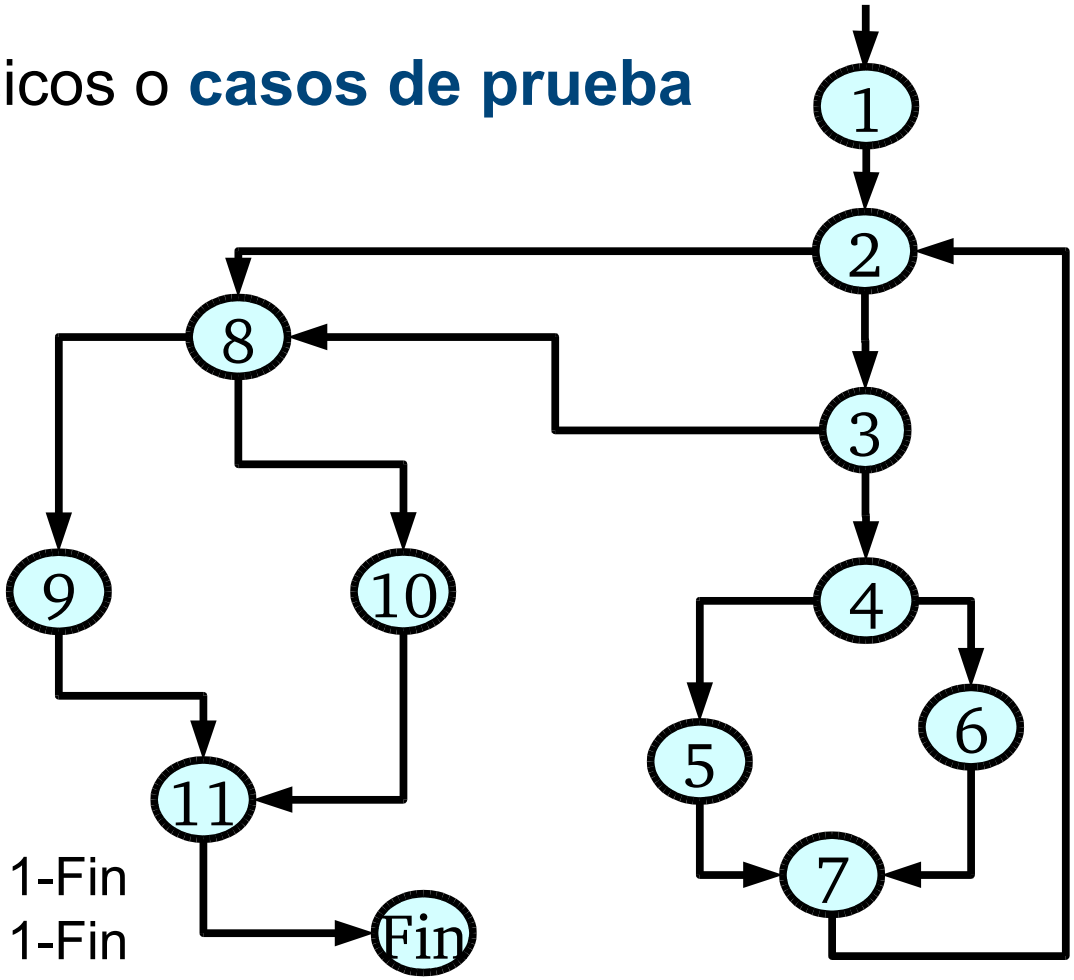
2. Calcular Complejidad del grafo

Complejidad = 5 (número de caminos básicos de ejecución)

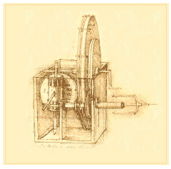


Técnica de caja blanca: El camino básico

3. Derivar caminos básicos o **casos de prueba**



Camino 1: 1-2-8-9-11-Fin
Camino 2: 1-2-8-10-11-Fin
Camino 3: 1-2-3-8-9-11-fin
Camino 4: 1-2-3-4-5-7-2-9-11-Fin
Camino 5: 1-2-3-4-6-7-2-9-11-Fin



Técnica de caja blanca: El camino básico

4. Derivar **datos de prueba** que cumplan con lo especificado en los casos de prueba (caminos básicos).

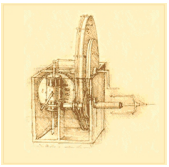
Camino 1: 1-2-8-9-11-Fin
Valores de entrada: secuencia = {} y valor=1
Valor esperado: resultado = -1

Camino 2: 1-2-8-10-11-Fin
No se puede probar

Camino 3: 1-2-3-8-9-11-fin
No se puede probar

Camino 4: 1-2-3-4-5-7-2-8-9-11-Fin
Datos de entrada : secuencia = {2} y valor = 1
Resultado esperado: resultado = -1

Camino 5: 1-2-3-4-6-7-2-8-9-11-Fin
Datos de entrada: secuencia = {1} y valor = 2
Resultado esperado: resultado = -1



Técnica de caja blanca: El camino básico

5. Verificar arista que no se han podido pasar o líneas de código sin ejecutar.

Camino 2: 1-2-8-**10**-11-Fin
Camino 3: 1-2-3-**8**-9-11-fin

Aristas que no se ha podido probar

Modificamos, por ejemplo, el camino 3 para forzar su paso por el nodo 8 y 10

Camino 3: 1-2-3-**8-10**-11-Fin
Datos de entrada: secuencia = {1} y valor = 1
Resultado esperado: resultado = 0