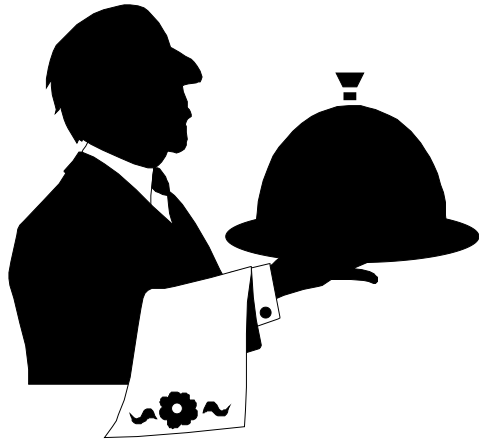


# Prueba de programas

## Programación I

Grado en Ingeniería Informática  
MDR, JCRdP y JDGD

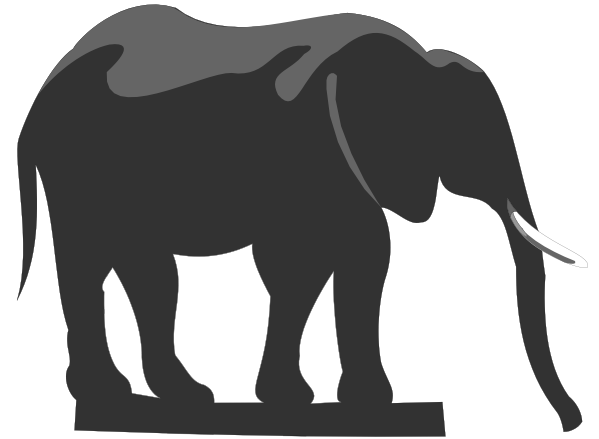
# Propiedades del software



Corrección



Amigabilidad



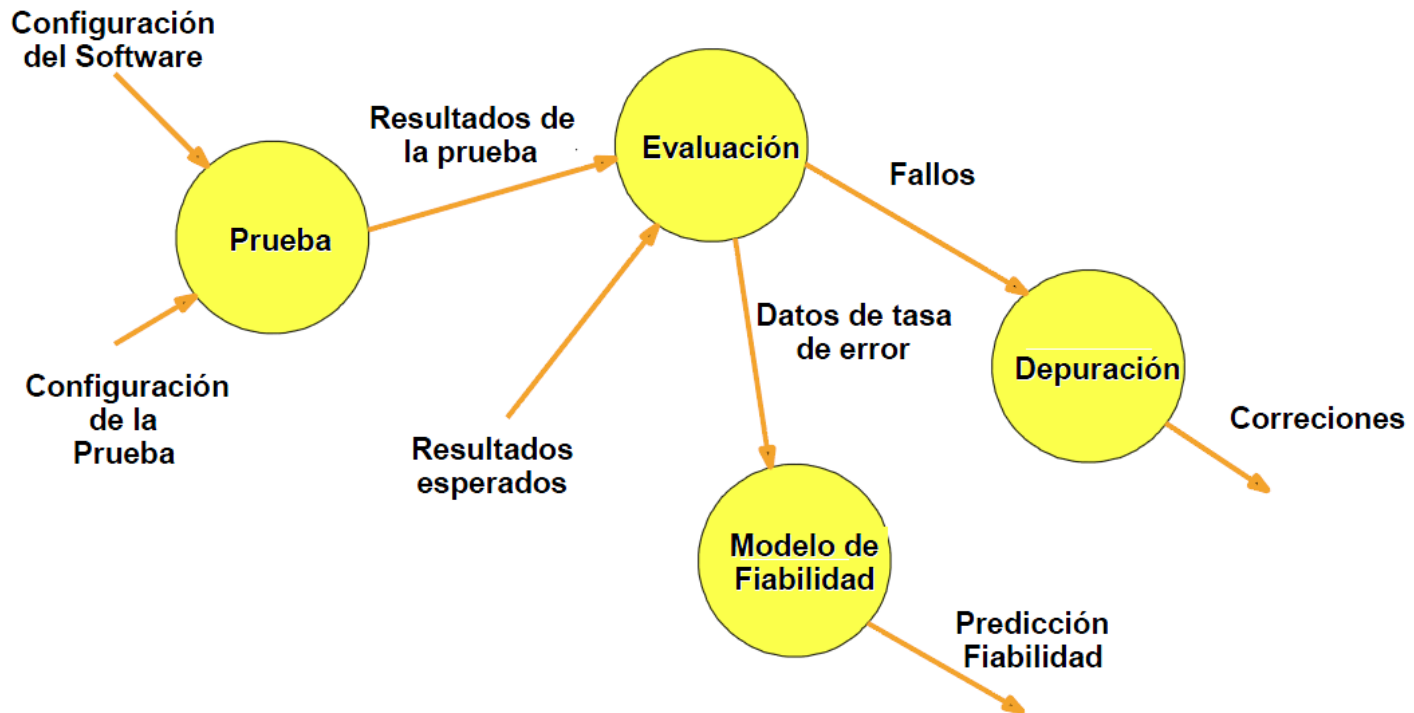
Robustez

# Introducción

- ▶ Las posibilidades de que se cometan **errores** en el desarrollo de programas son **enormes**
  - Encontrar estos errores es una tarea primordial
- ▶ La **correctitud** de un programa nunca estará garantizada si no se realiza una *verificación formal*

# Introducción

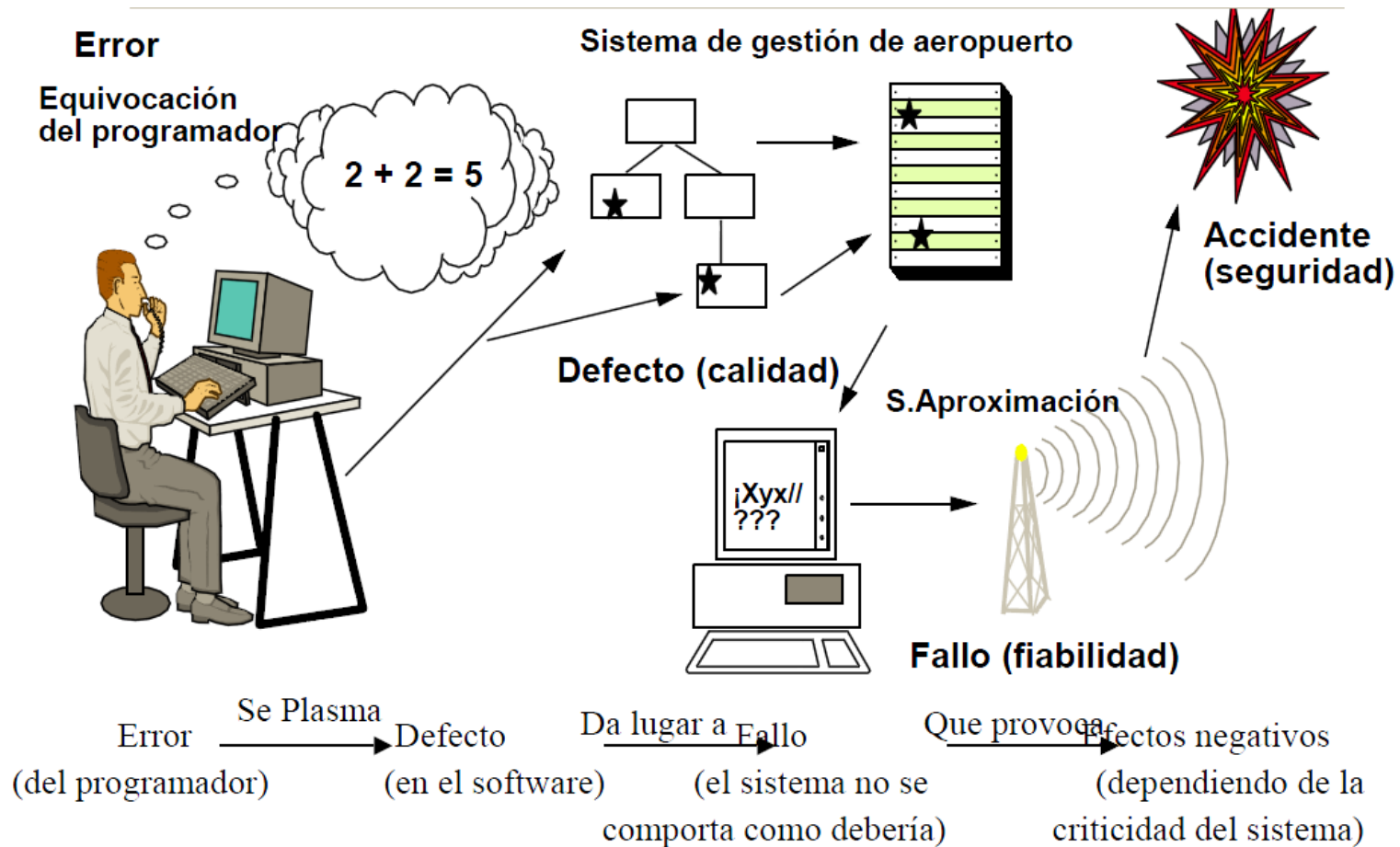
- ▶ Las **pruebas de software** son los procesos de **búsqueda de errores** en el software
  - Factor crítico para determinar la calidad del software



# Introducción

- ▶ El proceso de prueba consiste en comparar el estado o *comportamiento del software* con respecto a un *oráculo* (verdad preestablecida)
  - Este **oráculo** son las especificaciones, productos comparables, versiones previas del mismo producto, inferencias con respecto al propósito esperado, expectativas del cliente o usuario, estándares, etc.

# Errores, defectos y fallos

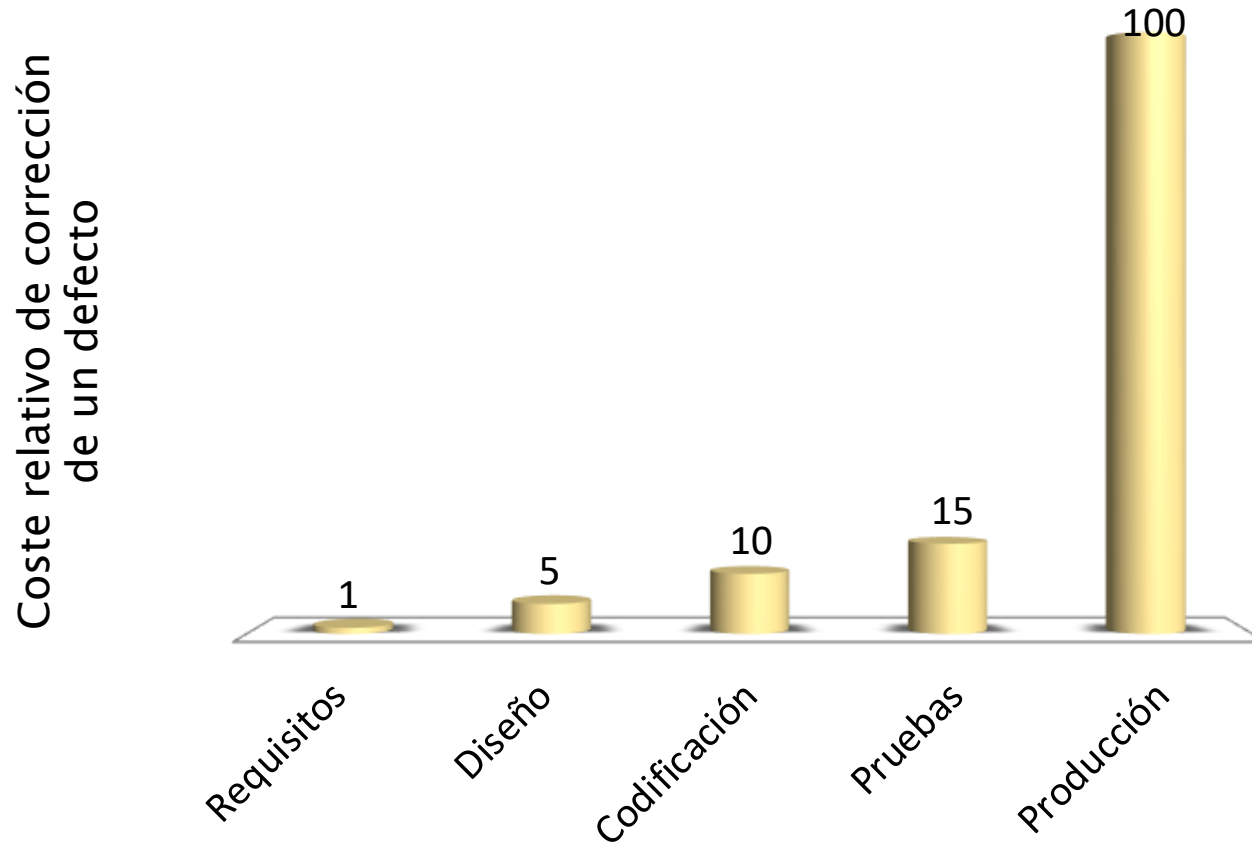


# Errores, defectos y fallos

- ▶ Los **defectos** en los programas son la plasmación de errores en el código, en los requisitos, etc.
- ▶ Los **fallos** son resultado de la ejecución de un defecto
  - Pueden existir defectos que no resultan en fallos, como código que no se ejecuta nunca
  - Otros defectos pueden mostrarse sólo al cambiar de plataforma (S.O. o máquina) o cambio de software con el que se interacciona

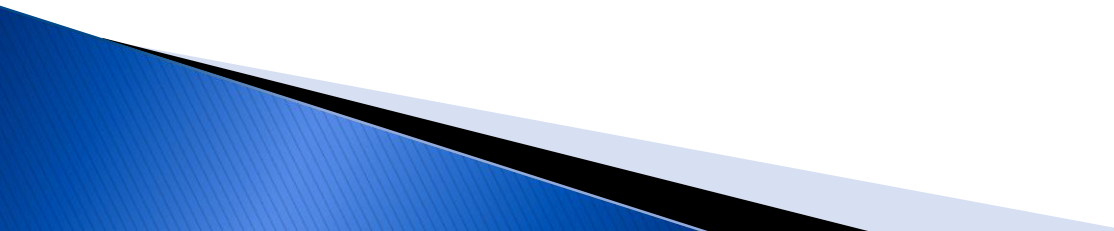
# Coste de los defectos

Barry Boehm, "Equity Keynote Address" March 19, 2007.





# Fundamento de las pruebas

- ▶ El **objetivo** de las pruebas es detectar los defectos del software con el menor coste y en el menor tiempo posible
  - ▶ Una prueba tiene éxito cuando encuentra un error
  - ▶ Una prueba se considera mejor que otra cuando tiene más posibilidades de encontrar un error
- 

# Fundamento de las pruebas

- ▶ Las pruebas **no pueden demostrar** la ausencia de defectos, sólo pueden señalar su existencia
  - Las pruebas, si se superan, solo muestran que el software funciona bien para los casos probados
- ▶ Las pruebas exhaustivas son impracticables
  - El número de entradas posibles a un programa hace inviable este cometido

# Características de las pruebas

- ▶ Las pruebas tienen más **probabilidad de éxito** si son realizadas por programadores que no desarrollaron el software (prejuicios).
- ▶ **Enfoque de las pruebas:**
  - Descubrir que el software no hace lo que debe hacer
  - Descubrir que el software hace lo que no debe hacer
- ▶ Las pruebas no deben dejar de explorar casos por suponerlos correctos a priori

# Características de las pruebas

- ▶ Las pruebas deben realizarse en modo ascendente
  - Inicialmente se deben centrar en elementos básicos e independientes como procedimientos, funciones y módulos
  - Después se deben comprobar los que usan los ya probados e ir subiendo el nivel de complejidad para finalizar con el programa completo
- ▶ Una buena prueba no debe ser redundante
  - El tiempo es un recurso importante y no hay motivo para repetir el mismo caso dos veces en una prueba
  - Cada prueba debe tener un objetivo distinto

# Casos de prueba

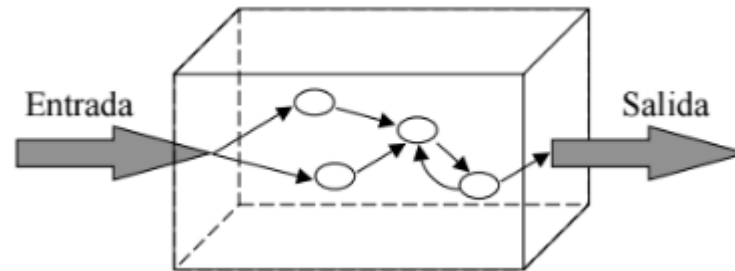
- ▶ Las pruebas **se concretan** en la ejecución de una serie de casos de prueba (batería de pruebas)
  - Seleccionados para encontrar errores con mayor probabilidad
- ▶ Cada **caso de prueba** se compone de una entrada de datos y secuencia de ejecución y unos datos esperados de salida (oráculo)
- ▶ Para **probar un caso**, se ejecutan las instrucciones para los datos de entrada, y se compara el resultado obtenido con el esperado
  - Se deben incluir **datos de entrada válidos** y esperados, y **no válidos** e inesperados

# Proceso de pruebas

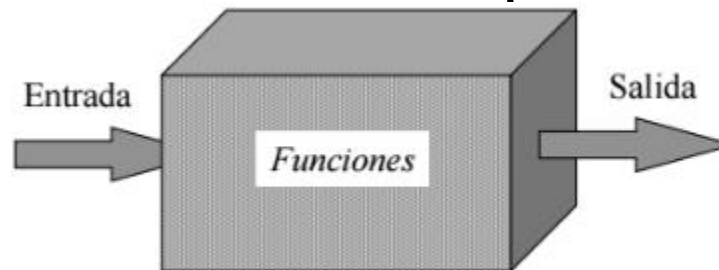
- ▶ Desarrollo de software tradicional:
  - Equipo de prueba independiente
  - Puede iniciarse cuando existan funcionalidades disponibles o al inicio del desarrollo
  - Fases: planificación, desarrollo de pruebas, ejecución, informe de errores, análisis de errores, reejecución, prueba de regresión, informe final
- ▶ Desarrollo de software extremo:
  - Diseño dirigido por pruebas (TDD)
  - Las pruebas son lo más importante
  - Primero se escriben las pruebas y después el código que las pasa

# Tipos de pruebas

- ▶ **Caja blanca:** Se basa en el conocimiento del código del programa a probar (cobertura de código)



- ▶ **Caja negra:** Se basa en la especificación (interfaz)



- ▶ **Caja gris:** Conociendo el código para mejorar los casos de prueba de caja negra

# Pruebas de caja blanca

## ► Objetivo:

- Probar todos los caminos por los que pasa el flujo de ejecución de un programa
- El número de caminos aumenta exponencialmente con el número de condiciones y bucles

## ► Prueba del Camino Básico:

- Se obtiene el grafo del flujo de ejecución del programa
- Se calcula la complejidad ciclomática
- Se obtiene el conjunto de caminos básicos
- Se diseñan los casos de prueba que fuerzan la ejecución de cada camino



# Grafo del flujo de ejecución

- ▶ El **grafo del flujo** representa los caminos de ejecución que se pueden seguir en un subprograma, módulo, etc.
- ▶ Es un **grafo orientado**
  - Los **vértices** representan instrucciones o conjuntos de instrucciones que se ejecutan como una unidad o condiciones en el flujo (*vértices predicados*)
  - Los **arcos** representan la posibilidad de que una vez terminada la ejecución de un vértice se pase a ejecutar otro

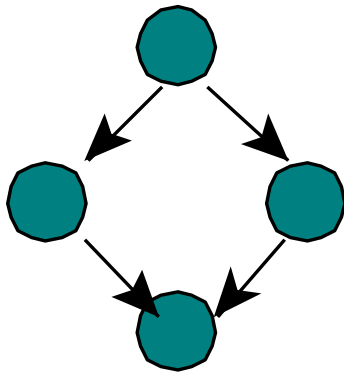
# Grafo del flujo de ejecución

Grafos de flujo de las sentencias de control

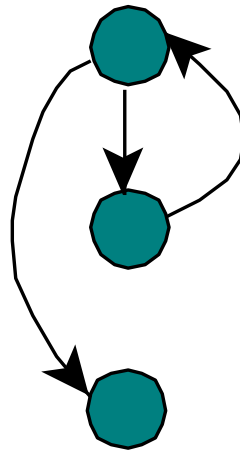
Secuencial



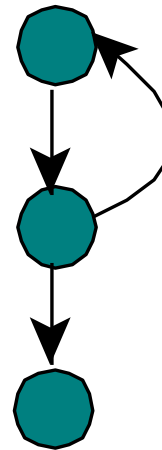
Alternativa



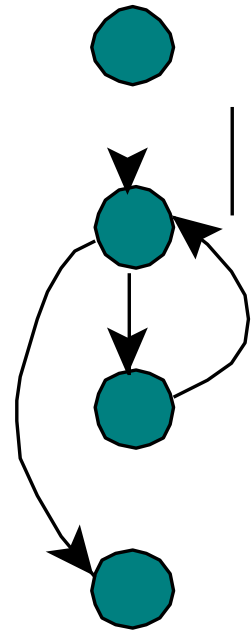
Mientras



Repetir hasta

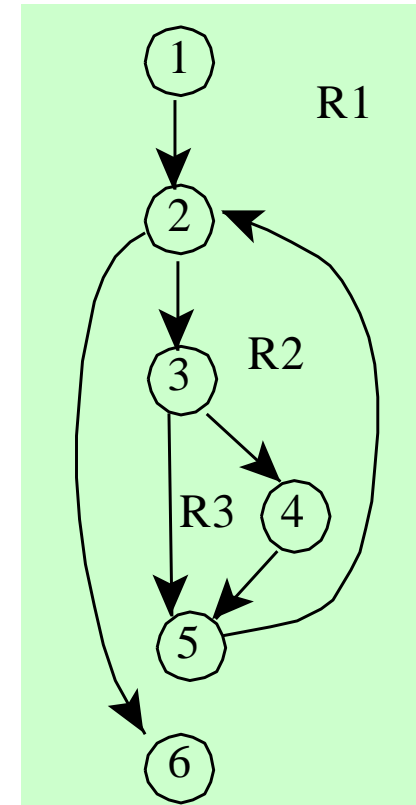


Para



# Ejemplo: grafo del flujo de la función MultiplicaciónRusa

```
public static int multRusa(int n, int m){  
    int a, b, p;  
1   a=n; b=m; p=0;  
2   while ( a > 0 ){  
3       if ( a%2 == 1 ){  
4           p = p+b;  
        }  
5       a=a/2; b=b*2;  
    }  
6   return p;  
}
```



# Complejidad ciclomática

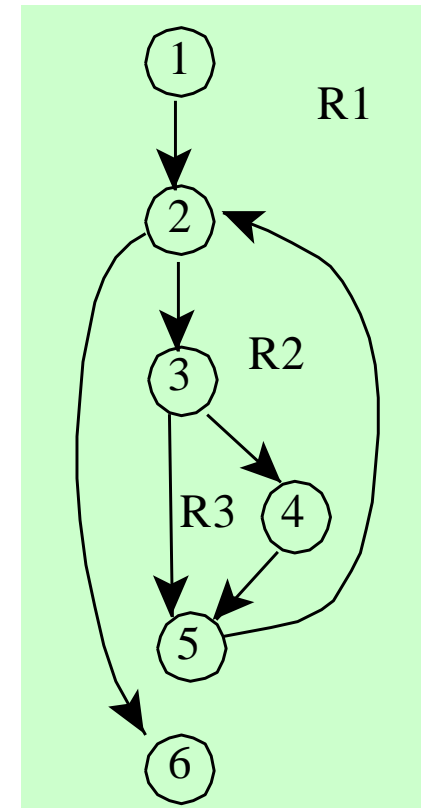
- ▶ Representa la complejidad de la lógica de ejecución del programa
- ▶ Determina el **número de caminos independientes** que forman un conjunto de caminos básicos
  - **Caminos independientes** son aquellos que al menos recorren un nuevo arco no utilizado en los caminos anteriores
  - Indica el **número mínimo de pruebas** que garantiza la ejecución de cada instrucción al menos una vez

# Complejidad ciclomática

- ▶ Sea  $G = \langle N, A \rangle$  el grafo del flujo,  $N$  el conjunto de vértices y  $A$  el de arcos
- ▶ La **complejidad ciclomática** se representa por  $CC(G)$  y se puede calcular de varias formas:
  - $CC(G)$  = Número de regiones en que se subdivide el plano que representa el grafo. La zona que queda fuera del grafo se considera una región más
  - $CC(G) = A - N + 2$
  - $CC(G) = V + 1$ , donde  $V$  es el número de vértices prediados (desde donde parte más de un arco)

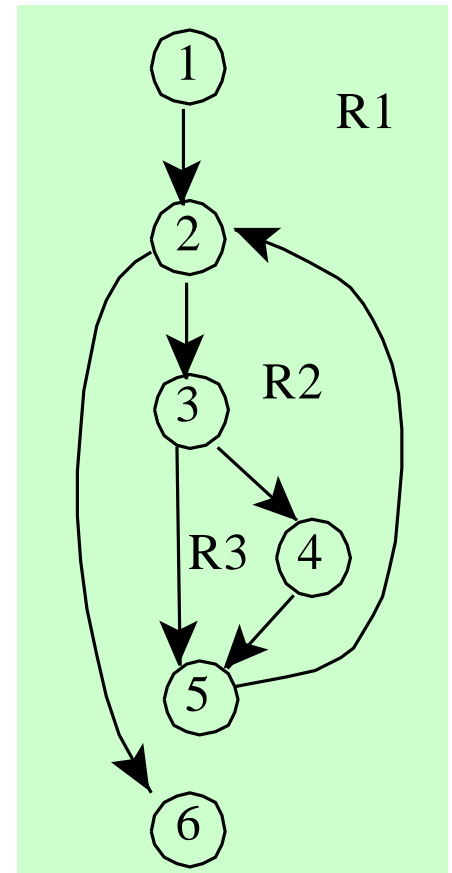
# Ejemplo: complejidad ciclomática de la función MultiplicaciónRusa

- La complejidad ciclomática según los distintos métodos:
  - $CC(G) = 3$ , porque existen las regiones R1, R2 y R3
  - $CC(G) = 7 - 6 + 2 = 3$ , debido a que  $A=7$  y  $N=6$
  - $CC(G) = 2 + 1$ , ya que de los vértices 2 y 3 son vértices predicados (sale más de un arco)



# Ejemplo: Casos de prueba para la función MultiplicaciónRusa

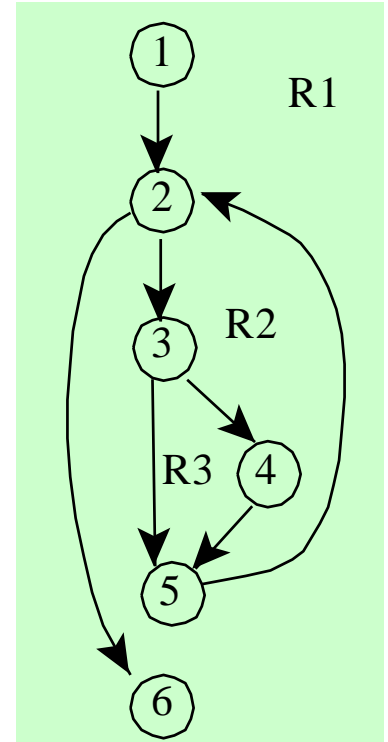
- Partiendo del código fuente se obtiene el grafo de flujo
- La complejidad ciclomática del grafo es 3
- El conjunto de caminos básicos es:
  - 1, 2, 6
  - 1, 2, 3, 5, 2, 6
  - 1, 2, 3, 4, 5, 2, 6



# Ejemplo: Casos de prueba para la función MultiplicaciónRusa

- Para el camino "1, 2, 6",  $n=0$  y  $m$  cualquier valor entero positivo. Resultado:  $p=0$

```
public static int multRusa(int n, int m){  
    int a, b, p;  
1   a=n; b=m; p=0;  
2   while ( a > 0 ){  
3       if ( a%2 == 1 ){  
4           p = p+b;  
        }  
5       a=a/2; b=b*2;  
    }  
6   return p;  
}
```

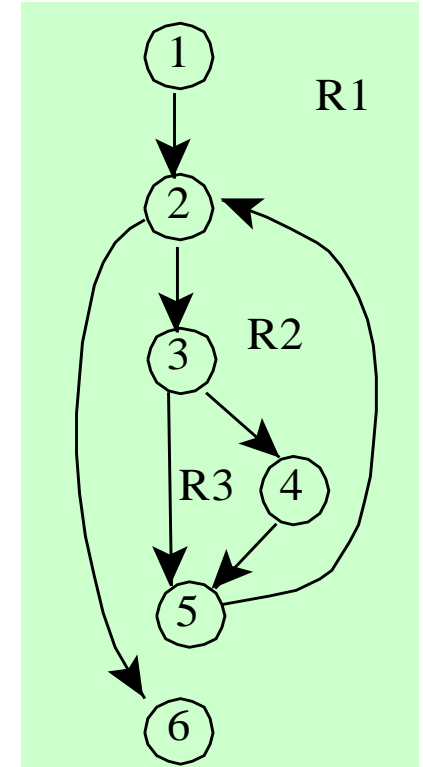




# Ejemplo: Casos de prueba para la función MultiplicaciónRusa

- Para el camino "1, 2, 3, 4, 5, 2, 6",  $n=1$  y  $m$  cualquier valor entero positivo. Resultado:  $p=m$

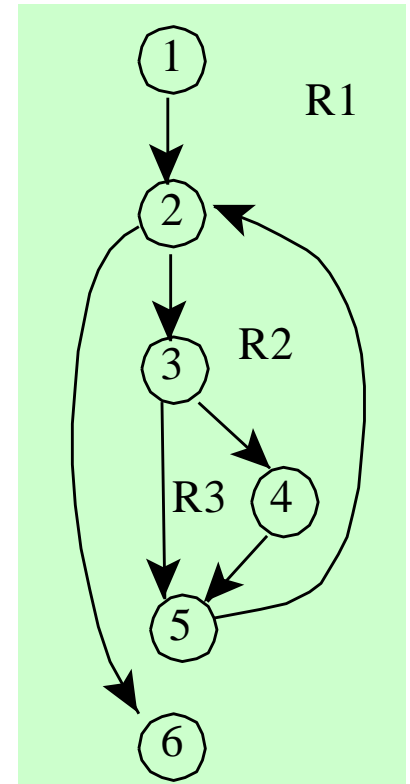
```
public static int multRusa(int n, int m){  
    int a, b, p;  
1   a=n; b=m; p=0;  
2   while ( a > 0 ){  
3       if ( a%2 == 1 ){  
4           p = p+b;  
        }  
5       a=a/2; b=b*2;  
    }  
6   return p;  
}
```



# Ejemplo: Casos de prueba para la función MultiplicaciónRusa

- Para el camino "1, 2, 3, 5, 2, 6",  $n=2$  y  $m$  cualquier valor entero positivo. Resultado:  $p=2*m$

```
public static int multRusa(int n, int m){  
    int a, b, p;  
1   a=n; b=m; p=0;  
2   while ( a > 0 ){  
3       if ( a%2 == 1 ){  
4           p = p+b;  
        }  
5       a=a/2; b=b*2;  
    }  
6   return p;  
}
```



# Pruebas de caja negra

## ► Objetivo:

- Encontrar las entradas cuya probabilidad de causar un fallo sea lo más alto posible
- Se pretende comprobar que la funcionalidad del software es la especificada
- El estudio de todas las posibles entradas y salidas es impracticable

## ► Técnicas:

- Clases de equivalencia
- Casos de prueba de los valores límites

# Clases de equivalencia

- ▶ Técnica de pruebas que divide las posibles entradas de un subprograma en particiones (subconjuntos disjuntos) de los que se pueden derivar los casos de prueba
- ▶ Las **particiones** se forman reuniendo las entradas para las que el subprograma tiene un comportamiento equivalente
- ▶ Los casos de prueba se diseñan de forma que **se pruebe** al menos un elemento de cada partición

# Clases de equivalencia

- ▶ Dos **entradas** son **equivalentes** si al ejecutarse desencadenan la misma secuencia de instrucciones
- ▶ Tipos:
  - Clases de equivalencia **válidas**: representan valores de entrada/salida válidos al programa
  - Clases de equivalencia **no válidas**: representan valores de entrada/salida no válidos al programa

# Límites del uso de la complejidad ciclométrica

- ▶ Dependiente del código
- ▶ Incompatible con la idea de "primero las prueba y después el código"
- ▶ No es aplicable a caja negra
- ▶ Sí aplicable a **caja gris**
  
- ▶ **Alternativa:** preestablecer las clases de equivalencia según las entradas y el "comportamiento esperado"
- ▶ **Se debe establecer una clase de equivalencia para cada tipo de caso que puede tener un comportamiento particular**

# Ejemplo: Diseño de casos para probar subprogramas

- ▶ Preferible que la ejecución dependa sólo de los datos de entrada (por diseño)
- ▶ Probar clases de equivalencia válidas y no válidas
- ▶ Probar para cada parámetro algunos valores generales y los **valores límite y adyacentes**

# Casos para un subprograma (1 / 2)

- ▶ Se desea generar **casos de prueba** para una función que comprueba si un vector de enteros está ordenado, o no
  - Tenemos como parámetro de entrada un vector y como salida un lógico
  - Los valores de entrada son **parametrizables** en dos aspectos: el tamaño del vector y los valores que contiene
  - La salida tiene dos valores (verdadero/falso)



# Casos para un subprograma (2 / 2)

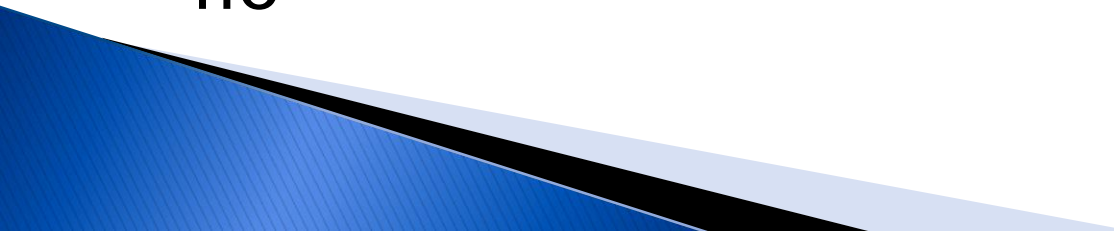
- ▶ **Tamaño:**
  - Extremos: 0, 1, 2, 3
  - Generales: varios valores adecuadamente grandes
- ▶ **Datos ordenados (resultado verdadero):**
  - Extremos: todos los valores iguales, ordenado con valores consecutivos, ordenado con valores consecutivos y/o iguales
  - Generales: varios casos con vectores ordenados con valores diversos
- ▶ **Datos ordenados (resultado falso):**
  - Extremos: con un solo valor desordenado al principio y al final del vector
  - Generales: varios casos con vectores desordenados con valores diversos

# Diseño de casos para probar clases

- ▶ Se generan casos para los distintos métodos (como en los subprogramas)
- ▶ Se debe tener en cuenta el estado de los objetos
  - El estado del objeto antes de la llamada es parte de los parámetros
  - El estado del objeto después de la llamada es parte del resultado del método
- ▶ Los distintos casos de cada método se deben combinar con los estados de los objetos
- ▶ No se debe prejuzgar el orden de llamada de los distintos métodos
- ▶ Los **errores** se encuentran frecuentemente en los **estados extremos**

# Ejemplo: Casos para una clase

## (1 / 2)

- ▶ Se tiene un contenedor que no admite repetidos
  - ▶ Puede almacenar como máximo 10 elementos
  - ▶ Se tiene tres operaciones: insertar, buscar y extraer
  - ▶ Las operaciones tienen como parámetro un dato y devuelve si la operación tuvo éxito o no
- 

# Ejemplo: Casos para una clase

## (2/2)

- ▶ Valores de entrada: No están limitados
- ▶ La salida tiene solo dos valores (verdadero/falso) e influyen en los valores de entrada que deberán ser valores almacenados y no almacenados en el contenedor
- ▶ Los estados del contenedor son
  - Extremos: vacío, un elemento, lleno, lleno-1
  - General: varios tamaños intermedios
- ▶ Los casos serán combinación de estados y de valores que están y no están en el contenedor

# Métodos accesorios, transformadores y neutros

- ▶ Los métodos accesorios: clases de equivalencias determinadas por los distintos estados de los objetos
- ▶ Los neutros: clases de equivalencias determinadas por los distintos estados de los objetos y los de los parámetros y de los datos devueltos
- ▶ Los transformadores: clases de equivalencias determinadas por los distintos estados de los objetos y los de los parámetros y de los datos devueltos y estado final

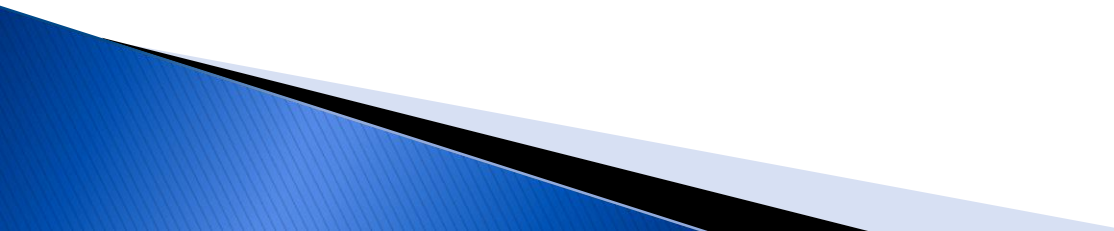
# Niveles de pruebas

- ▶ **Unidades:** Tiene como objetivo un módulo (clase) del código
- ▶ **Integración:** Encontrar fallos al unir los módulos de un programa
- ▶ **Sistemas:** Funcionamiento de un sistema completo en máquinas especificadas
  - **Alpha:** Pruebas de usuarios/clientes finales simulados o grupos reducidos. Software aún en desarrollo (incompleto)
  - **Beta:** Después de las Alpha se suministra o libera a conjuntos reducidos de usuarios. Software funcional en pruebas
  - **RC (Release Candidate).** Después de la Beta, si no se encuentran errores es la versión definitiva

# Niveles de pruebas

- ▶ **Regresión:** Comprobar los efectos de una corrección de error con respecto a versión anterior
- ▶ **Aceptación:** Pruebas realizadas por el cliente para comprobar que el software hace lo que quería

# Herramientas para pruebas

- ▶ Software de ayuda a realización de pruebas (JUnit)
  - ▶ Depurador (gdb, jdb, ddd, etc.)
  - ▶ Monitores de ejecución (Valgrind)
  - ▶ Comprobadores de cobertura (gcov, cobertura)
  - ▶ Análisis de rendimiento (gprof)
- 



# Bibliografía

- ▶ Java tools for Extreme Programming: mastering open source tools including Ant, JUnit, and Cactus / Richard Hightower, Nicholas Lesiecki, John Wiley & Sons, New York: (2002)
- ▶ **Pruebas de software**  
[http://es.wikipedia.org/wiki/Pruebas\\_de\\_software](http://es.wikipedia.org/wiki/Pruebas_de_software)