

## Sesión S04: Diseño de pruebas: **caja negra**



You are a lucky bug. I'm seeing that you'll  
be shipped with the next three releases

copyright 2005 Kazem A. Andekani

Diseño de casos de prueba: functional testing

- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de **comportamientos especificados**
- El conjunto de casos de prueba obtenido debe detectar el máximo número posible de defectos en el código, con el mínimo número posible de "filas" (eficacia y efectividad)

Método de **Particiones equivalentes**

- Paso 1: Análisis de la especificación: Particionamos cada cada entrada (y salida) en conjuntos "equivalentes"
- Paso 2: Selección de comportamientos usando las particiones obtenidas:
- Paso 3. Obtención del conjunto de casos de prueba

Ejemplos y ejercicios

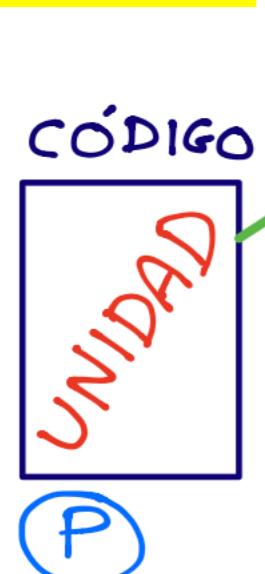
Vamos al laboratorio...

# DISEÑO DE CASOS DE PRUEBA

Seleccionamos de forma sistemática un conjunto de casos de prueba efectivo y eficiente!!

Hay DEFECTOS en el código?

DISEÑO

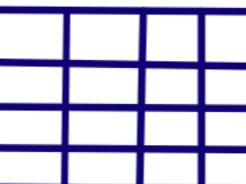


CFG  
camino básico

CC

caminos indep

TABLA



ESPECIFICACIÓN



particiones equivalentes

AUTOMATIZACIÓN

DRIVERS



ejecución HANEN JUnit

DETECTAR DEFECTOS (VERIFICACIÓN)



S/N



Para resolver el problema (responder a la pregunta inicial), necesariamente tenemos que ejecutar código: estamos haciendo pruebas DINÁMICAS!!

Independientemente del método de diseño elegido, SIEMPRE obtendremos un conjunto EFICIENTE y EFECTIVO

Ya hemos visto una forma de seleccionar los comportamientos a probar a partir del código implementado (métodos estructurales)

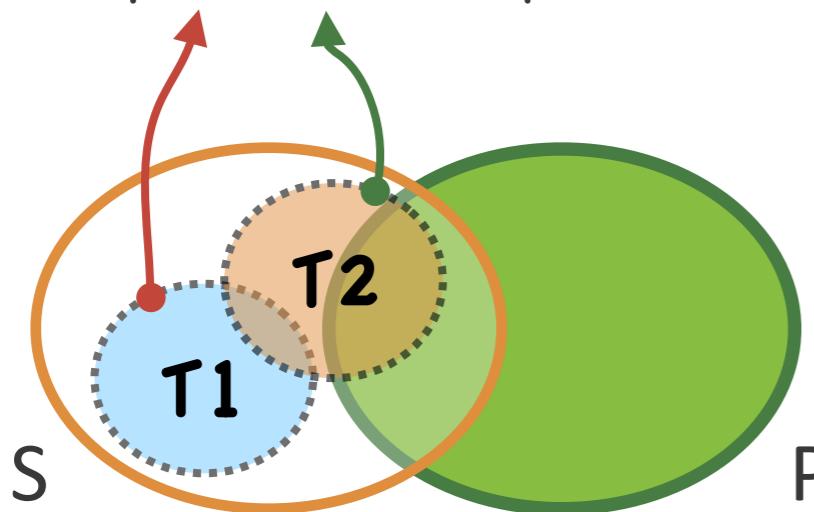
Ahora vamos a explicar cómo hacerlo partiendo de la ESPECIFICACIÓN.

AMBAS técnicas son necesarias y complementarias

# FORMAS DE IDENTIFICAR LOS CASOS DE PRUEBA

## FUNCTIONAL TESTING

Comportamientos probados



Podemos detectar comportamientos NO IMPLEMENTADOS

Nunca podremos detectar comportamientos implementados, pero no especificados

- Cualquier programa puede considerarse como una función que “mapea” valores desde un dominio de entrada a valores en un dominio de salida. El elemento a probar se considera como una “**caja negra**”
- Los casos de prueba obtenidos son independientes de la implementación
- El diseño de los casos de prueba puede realizarse en paralelo o antes de la implementación



Los métodos de diseño basados en la ESPECIFICACIÓN:

1. **Analizan** la especificación y **PARTICIONAN** el conjunto S (dependiendo del método se puede usar una representación en forma de grafo)
2. **Seleccionan** un conjunto de **comportamientos** según algún criterio
3. **Obtienen** un conjunto de **casos de prueba** que ejercitan dichos comportamientos

# MÉTODOS DE DISEÑO DE CAJA NEGRA

- P
- Existen MUCHOS métodos de diseño de pruebas de caja negra:

- Método de particiones equivalentes
- Método de análisis de valores límite
- Método de tablas de decisión

Pruebas unitarias

- Método de grafos causa-efecto
- Método de diagramas de transición de estados
- Método de pruebas basado en casos de uso

Pruebas de sistema

- Método de pruebas basado en requerimientos
- Método de pruebas basado en escenarios

Pruebas de aceptación

En todos ellos, la identificación de DOMINIOS de entradas y salidas contribuye a PARTICIONAR los comportamientos en clases (particiones)

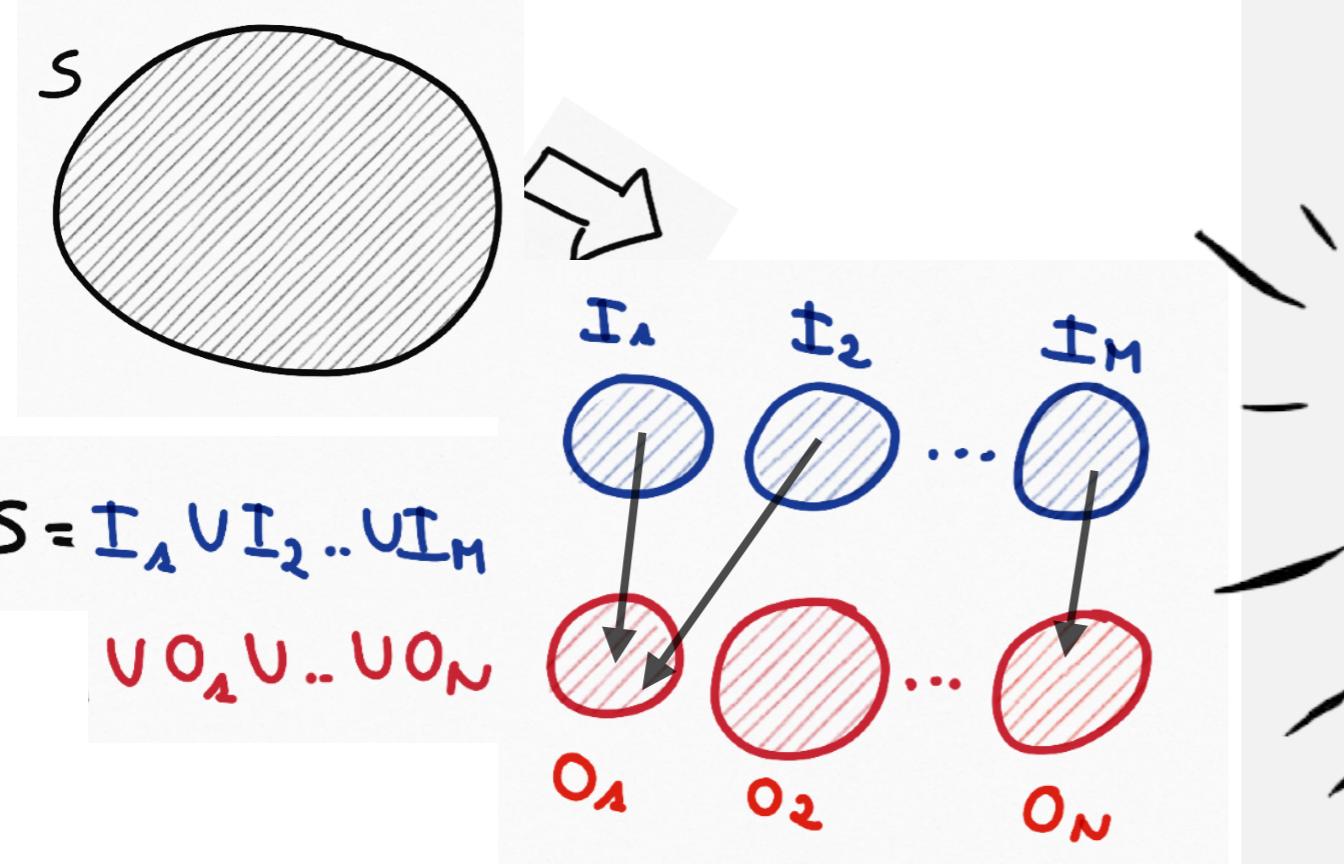
A diferencia de los métodos de caja blanca, se pueden aplicar en CUALQUIER nivel de pruebas

- Se trata de un proceso SISTEMÁTICO que identifica, a partir de la ESPECIFICACIÓN disponible, un conjunto de CLASES de equivalencia para **cada** una de las **entradas** y **salidas** del "elemento" (unidad, componente, sistema) a probar

- Cada clase de equivalencia (o partición) de entrada representa un subconjunto del total de datos posibles de entrada que tienen un mismo comportamiento (Los elementos de una partición de entrada se caracterizan por tener su "imagen" en la misma partición de salida)

## MÉTODO DE DISEÑO: PARTICIONES EQUIVALENTES

Sesión 4: Diseño de pruebas de Caja Negra



P

**S**

El **OBJETIVO** es **MINIMIZAR** el número de casos de prueba requeridos para cubrir **TODAS** las particiones al menos una vez, teniendo en cuenta que las particiones de entrada inválidas se prueban de una en una.

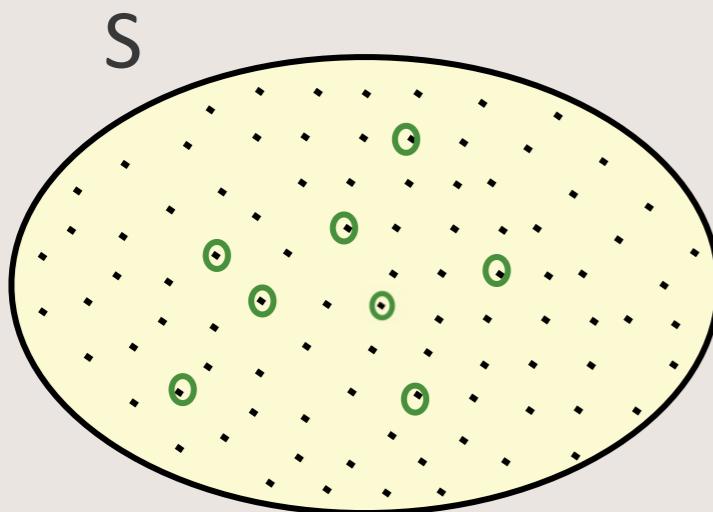
- Cada caso de prueba usará un **subconjunto** de particiones
- **NO** se trata de probar **TODAS** las combinaciones posibles, sino de garantizar que **TODAS** las particiones de entrada (y de salida) se prueban **AL MENOS UNA VEZ**

# SISTEMATICIDAD Y PARTICIONAMIENTOS

detectar el máximo nº posible de errores

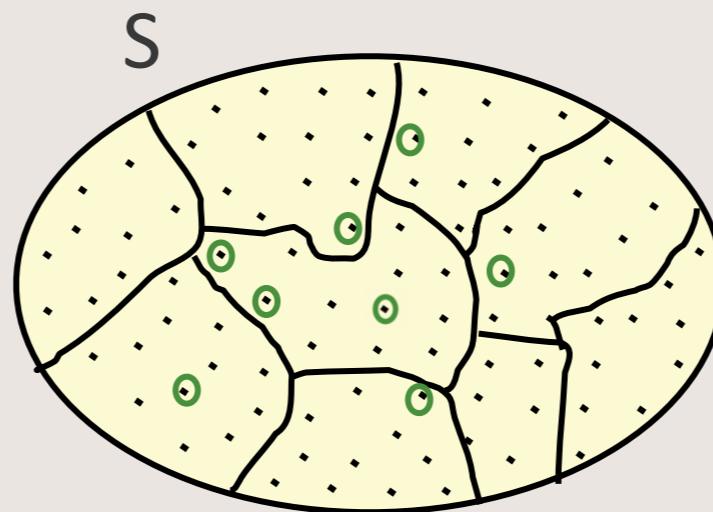
... con el MENOR nº posible de casos de prueba

- P Para conseguir un conjunto de pruebas EFECTIVO y EFICIENTE, tenemos que ser SISTEMÁTICOS a la hora de determinar las particiones de entrada/ salida
  - P Las particiones representan conjuntos de posibles comportamientos del sistema
  - P Se deben elegir muestras significativas de CADA partición
  - P Tenemos que asegurarnos de que cubrimos TODAS las particiones

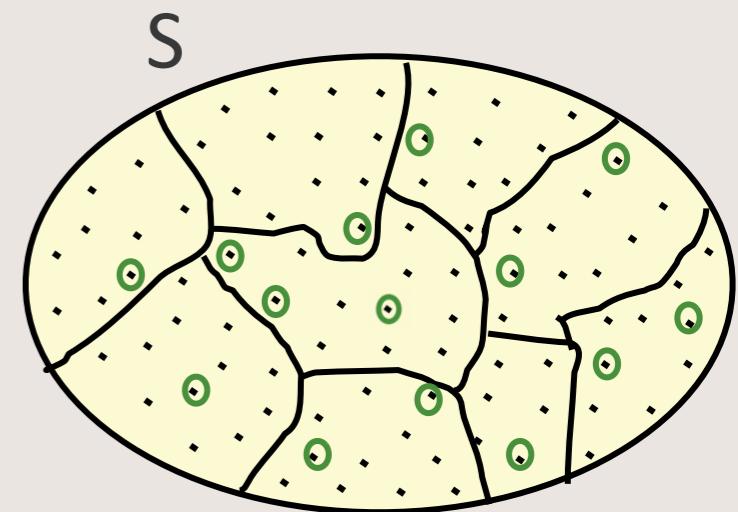


**No particiones.** Datos de prueba (círculos verdes) elegidos aleatoriamente

Las pruebas no son efectivas (hay tipos de comportamientos sin probar) ni eficientes (hay datos de prueba redundantes)



**Particiones.** Se eligen muestras de cada partición



Aquí aseguramos la efectividad del diseño (probamos TODOS los tipos de comportamientos diferentes). Mantenemos algunos datos de prueba redundantes.

# ¿CÓMO IDENTIFICAMOS UNA PARTICIÓN?

Particionamos CADA ENTRADA

P Las particiones (o clases de equivalencia) se identifican en base a CONDICIONES de entrada/salida de la unidad a probar (de hecho en la literatura se utilizan indistintamente los términos partición de entrada, clase de equivalencia de entrada o condición de entrada)

P Una condición de entrada/salida, puede aplicarse a una única variable de entrada/salida en una especificación o con un subconjunto de ellas

□ P.ej. Dados tres enteros: a, b, c, que representan los lados de un triángulo con valores positivos menores o iguales a 20 ...

\* particiones de entrada:

- (1)  $a, b, c > 0$  y  $a, b, c \leq 20$
- (2)  $a > 20$
- (3)  $b > 20$
- (4)  $c > 20$
- ...

la condición de entrada se aplica a las variables a, b y c

la condición de entrada sólo se aplica a una variable



Lógicamente, para poder identificar las condiciones sobre las entradas, primero hay que tener claro cuántas y qué ENTRADAS tiene el elemento que queremos probar!!!!

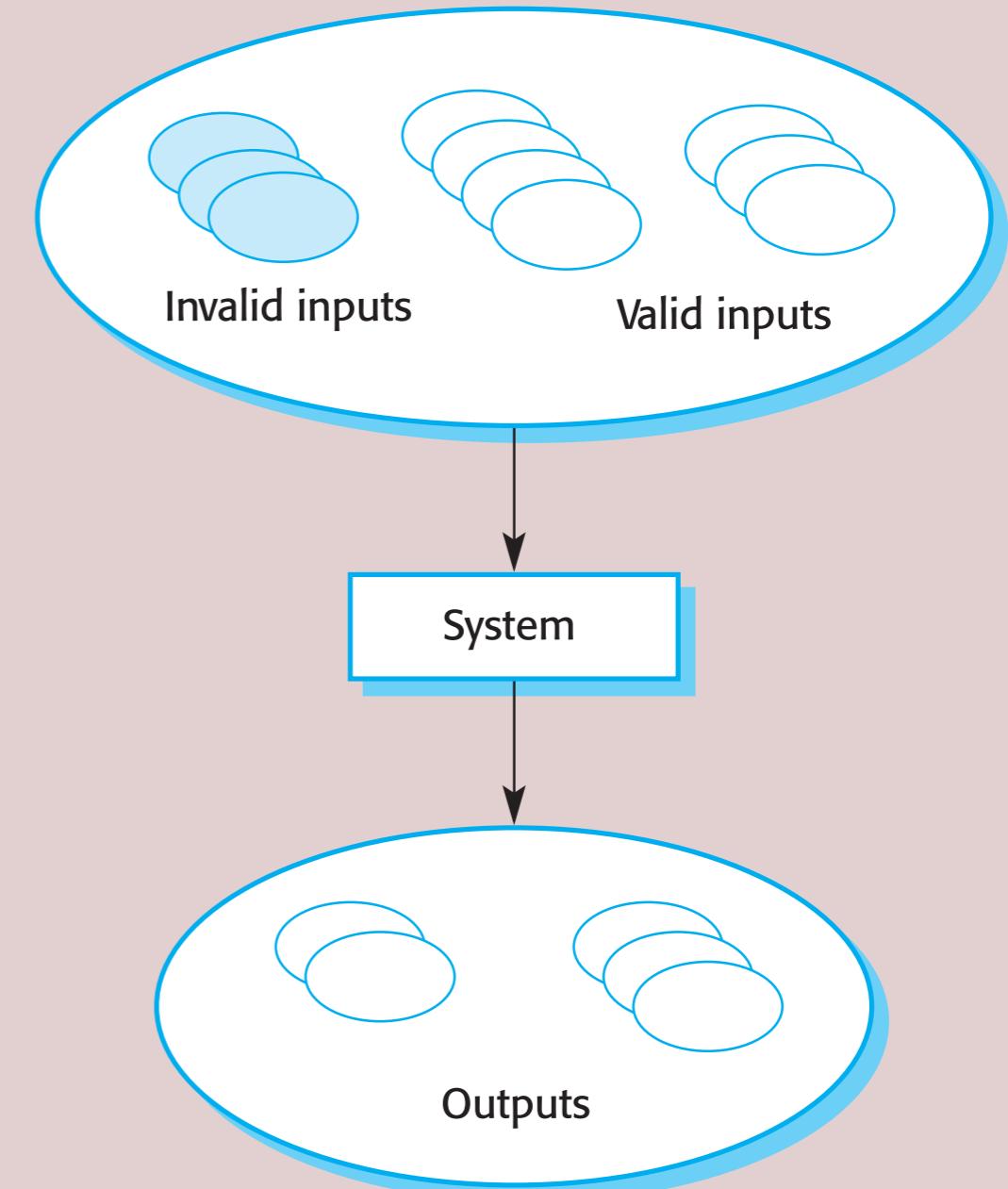
# MÁS SOBRE PARTICIONES DE ENTRADA/SALIDA

- Las "variables" de entrada/salida **no necesariamente** se corresponden con "parámetros" de entrada/salida de la unidad a probar
  - P.ej. El método `validar_pin()` comprueba si un pin (obtenido invocando a otra unidad) es válido o no. Si es válido, el método devuelve también el pin obtenido...
    - \* Supongamos que el método a probar es:
      - **boolean validar\_pin(Pin pinValido)**
    - \* Las "variables" de entrada/salida que debemos considerar son:
      - Entrada: tupla con un máximo de 3 "intentos", en donde cada intento = valor del pin obtenido por la unidad externa + código de respuesta devuelto por la unidad externa
      - Salida: booleano + objeto pin válido
- Las particiones deben ser DISJUNTAS (las particiones No comparten elementos).
- Recordad además que todos los miembros de una partición de entrada deben tener su "imagen" en la misma partición de salida (si dos elementos de la misma partición de entrada se corresponden con dos elementos de particiones de salida diferentes, entonces la partición de entrada NO está bien definida)

# PARTICIONES VÁLIDAS E INVÁLIDAS

P

- Las clases de equivalencia (condiciones, particiones) de entrada, pueden clasificarse como **VÁLIDAS** o **INVÁLIDAS**.
  - Ej: variable "mes" de tipo entero que representa un mes del año.
    - \* Clase válida: Los valores 1..12 son valores válidos.
    - \* Clases inválidas: Un valor superior a 12, o inferior a 1 podemos considerarlos inválidos.
- Las particiones de entrada inválidas normalmente tienen asociadas clases de salida inválidas.



Sólo puede haber una partición INVÁLIDA de entrada en un caso de prueba

# IDENTIFICACIÓN DE LAS CLASES DE EQUIVALENCIA



Debes usarlas  
SIEMPRE!!

P

P

**Paso 1.** Identificar las clases de equivalencia (particiones) para **CADA** entrada/salida (E/S), siguiendo las siguientes HEURÍSTICAS:

- #1 Si la E/S especifica un RANGO de valores válidos, definiremos una clase válida (dentro del rango) y dos inválidas (fuera de cada uno de los extremos del rango). Ej. x puede tomar valores entre 1..12. Clase válida:  $x = 1..12$ ; Clases inválidas:  $x > 12$  y  $x < 1$
- #2 Si la E/S especifica un NÚMERO N de valores válidos, definiremos una clase válida (número de valores entre 1 y N) y dos inválidas (ningún valor, más de N valores). Ej. x puede tomar entre 1 y 3 valores. Clase válida: x toma entre 1 y 3 valores; Clases inválidas: x no tiene ningún valor y x tiene más de 3 valores
- #3 Si la E/S especifica un CONJUNTO de valores válidos, definiremos una clase válida (valores pertenecientes al conjunto) y una inválida (valores que no pertenecen al conjunto). Ej. x puede ser uno de estos tres valores {valorA, valorB, valorC}. Clase válida: x toma uno de los valores  $\in$  al conjunto; Clase inválida: x toma cualquier valor que no  $\in$  al conjunto
- #4 Si por alguna razón, se piensa que cada uno de los valores de entrada se van a tratar de forma diferente por el programa, entonces definir una clase válida para cada valor de entrada
- #5 Si la E/S especifica una situación DEBE SER, definiremos una clase válida y una inválida. Ej. x comenzar por un número. Clase válida: x empieza con un dígito; Clase inválida: x NO empieza por un dígito
- #6 Si por alguna razón, se piensa que los elementos de una partición van a ser tratados de forma distinta, subdividir la partición en particiones más pequeñas

# IDENTIFICACIÓN DE LOS CASOS DE PRUEBA

## O Paso 2. Identificar los casos de prueba de la siguiente forma:



El orden  
de los  
pasos  
importa!!

Debemos asignar un IDENTIFICADOR ÚNICO para cada partición

**2.1** Hasta que todas las clases válidas estén cubiertas (probadas), escribir un nuevo caso de prueba que cubra el máximo número de clases válidas todavía no cubiertas

**2.2** Hasta que todas las clases inválidas estén cubiertas (probadas), escribir un nuevo caso de prueba que cubra una y sólo una clase inválida (de entrada) todavía no cubierta

Si un caso de prueba contiene más de una clase de entrada NO válida, puede que alguna de ellas no se ejecute nunca, ya que alguna de las clases no válidas puede "enmascarar" a alguna otra, o incluso terminar con la ejecución del caso de prueba

**2.3** Elegir un valor concreto para cada partición

El resultado de este proceso será una TABLA con tantas FILAS como CASOS de PRUEBA hayamos obtenido

- \* Cada caso de prueba "cubrirá" un subconjunto de las particiones
- \* El conjunto obtenido debe contemplarlas todas como mínimo una vez
- \* Sólo puede haber una partición inválida de entrada en un caso de prueba

# EJEMPLO 1: IMPRESIÓN DE CARÁCTERES

3 Entradas + 1 Salida

**ESPECIFICACIÓN:** Método en el que, dados como entradas: un carácter X introducido por el usuario, un número N entre 5 y 10, y el valor “rojo” o “azul”, devuelve (salida) una cadena de N caracteres X de color rojo o (N-1) caracteres de color azul, o bien el mensaje “ERROR: repite entrada” si el usuario proporciona un valor de N < 5 ó N >10.

- **Entrada 1 (E1) (carácter X):** puede ser cualquier carácter
  - Clase válida: V1
- **Entrada 2 (E2)(número N):** un valor comprendido entre 5 y 10
  - Clase válida: V2: valores entre 5 y 10 ( $5 \leq N \leq 10$ )
  - Clases inválidas: N1: valores menores que 5 ( $N < 5$ ), y N2: valores mayores que 10 ( $N > 10$ )
- **Entrada 3:** uno de los valores: “rojo”, “azul”
  - Clases válidas: V3: “rojo”, V4: “azul”
- **Salida (cadena de N caracteres):**
  - Clase válida: S1: Cadena de N caracteres de color rojo
  - Clase válida: S2: Cadena de (N-1) caracteres de color azul
  - Clase inválida: NS1: “ERROR: repite entrada”

Opcionalmente, podríamos haber añadido  
**N3:** entrada con más de 1 carácter.  
**NS2:** ?? (salida no especificada)  
 Pero E1 es de tipo "char", no son necesarias

**Nota:** nos indican que los valores “rojo” o “azul” se elegirán de una lista desplegable

Clases	Datos Entrada			Resultado Esperado
	E1	E2	E3	
V1-V2-V3-S1	‘c’	7	"rojo"	“ccccccc”
V1-V2-V4-S2	‘x’	6	"azul"	“xxxxx”
V1-N1-V4-NS1	‘c’	3	"azul"	“ERROR: repite entrada”
V1-N2-V4-NS1	‘j’	13	"azul"	“ERROR: repite entrada”

## EJEMPLO 2: VALIDAR FECHA

2 Entradas + 1 Salida

P

S

**ESPECIFICACIÓN:** El método `valida_fecha()` tiene como parámetros de entrada las variables de tipo entero: día y mes, de forma que dados ambos valores, devuelve cierto o falso, en función de que sea una fecha válida. Supongamos que el año es 2020

P

○ En este caso:

- para realizar las particiones aplicamos las condiciones de entrada al subconjunto formado por día y mes, ya que:
  - \* hay valores de entrada de una variable que pueden considerarse válidos o inválidos, dependiendo del valor de la otra variable. Por ejemplo el día 31, y los meses febrero y marzo
- por lo tanto consideraremos una única entrada:
  - \* (dia, mes) agrupamos las 2 entradas en una para realizar las particiones
- y como salida:
  - \* valor booleano indicando si la fecha es válida o no
- además, aplicaremos la regla #6, y subdividiremos tanto el día como el mes en particiones más pequeñas



**Regla #6:** Si por alguna razón, se piensa que los elementos de una partición van a ser tratados de forma distinta, subdividir la partición en particiones más pequeñas

Primero hay que identificar las E/S y luego las particionamos (SIEMPRE!!!)

# EJEMPLO 2: VALIDAR FECHA (PARTICIONES)

2 Entradas + 1 Salida

agrupamos las 2 en una única entrada  
para realizar las particiones

- Aplicamos las condiciones de entrada a las variables de entrada y salida, de forma que obtenemos las siguientes particiones:  
(Paso 1)

Particiones	
Entrada: dia(D) + mes(M)	Salida: S
DM1: d = {1..29} $\wedge$ m = {1..12}	S1: true
DM2: d = {30} $\wedge$ m = {1,3,..,12}	NS1: false
DM3: d = {31} $\wedge$ m = {1,3,5,7,8,10,12}	
NDM1: d > 31 $\wedge$ m = {1..12}	
NDM2: d < 1 $\wedge$ m = {1..12}	
NDM3: d = {30} $\wedge$ m = {2}	
NDM4: d = {31} $\wedge$ m = {2,4,6,9,11}	
NDM5: m > 12 $\wedge$ d = {1..31}	
NDM6: m < 1 $\wedge$ d = {1..31}	



Aplicamos la regla #6



Si agrupas entradas,  
NUNCA debes considerar más de una partición inválida en dicha agrupación.

ENTRADA: 3 particiones válidas + 6 particiones inválidas

SALIDA: 1 partición válida + 1 partición inválida

# EJEMPLO 2: TABLA RESULTANTE DE VALIDA\_FECHA() (PASO 2)

- Una posible elección de casos de prueba podría ser ésta:

(Paso 2)

Particiones	dia	mes	salida
DM1-S1	14	5	true
DM2-S1	30	6	true
DM3-S1	31	7	true
NDM1-NS1	43	10	false
NDM2-NS1	-3	6	false
NDM3-NS1	30	2	false
NDM4-NS1	31	4	false
NDM5-NS1	29	16	false
NDM6-NS1	29	-3	false

siempre valores CONCRETOS!!!!



# EJEMPLO 3: EL PROBLEMA DEL TRIÁNGULO

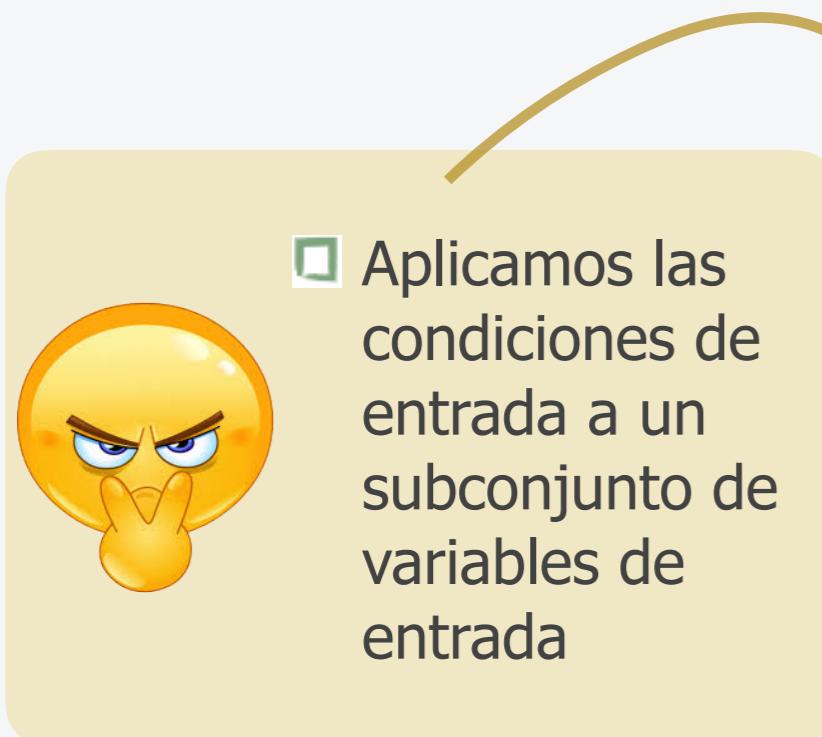
3 Entradas + 1 Salida

agrupamos las 3 en una única entrada  
para realizar las particiones

**ESPECIFICACIÓN:** dados tres enteros:  $a$ ,  $b$ , y  $c$ , que representan la longitud de los lados de un triángulo: cada uno de ellos debe tener un valor positivo menor o igual a 20. La unidad a probar, a partir de las entradas  $a$ ,  $b$  y  $c$  devuelve el tipo de triángulo:

- \* "Equilátero", si  $a = b = c$
- \* "Isósceles", si dos cualesquiera de sus lados son iguales y el tercero desigual
- \* "Escaleno", si dos cualesquiera de sus lados son desiguales
- \* "No es un triángulo", si  $a \geq b+c$ ,  $b \geq a+c$ , ó  $c \geq a+b$

○ Paso 1. Inicialmente podemos definir las siguientes particiones de entrada:



Aplicamos las condiciones de entrada a un subconjunto de variables de entrada

Entrada: $(a,b,c)$	
$C_1: (a,b,c > 0) \wedge (a,b,c \leq 20)$	$NC_1 = a > 20$
	$NC_2 = b > 20$
	$NC_3 = c > 20$
	$NC_4 = a \leq 0$
	$NC_5 = b \leq 0$
	$NC_6 = c \leq 0$

# EJEMPLO 3: PARTICIONES

P

S

ENTRADA: 4 particiones válidas +  
6 particiones inválidas

- Utilizando la **heurística #6** del Paso 1, vamos a dividir C1 en subclases, puesto que diferentes combinaciones de valores de a,b, y c se van a tratar de forma diferente (darán lugar a diferentes salidas):
  - es-triángulo:  $a < b+c \wedge b < a+c \wedge c < a+b$
- También particionamos las salidas

SALIDA: 4 particiones válidas +  
1 partición inválida

Entrada: a,b,c	Salida	
C <sub>11</sub> : a=b=c	NC <sub>1</sub> = a > 20	S <sub>1</sub> : "Equilátero"
C <sub>12</sub> : (a=b $\wedge$ a <> c) $\vee$ (a=c $\wedge$ a <> b) $\vee$ (b=c $\wedge$ b <> a)	NC <sub>2</sub> = b > 20	S <sub>2</sub> : "Isósceles"
C <sub>13</sub> : (a <> b) $\wedge$ (a <> c) $\wedge$ (b <> c)	NC <sub>3</sub> = c > 20	S <sub>3</sub> : "Escaleno"
C <sub>14</sub> : (a $\geq$ b+c) $\vee$ (b $\geq$ a+c) $\vee$ (c $\geq$ a+b)	NC <sub>4</sub> = a $\leq$ 0 NC <sub>5</sub> = b $\leq$ 0 NC <sub>6</sub> = c $\leq$ 0	S <sub>4</sub> : "No es triángulo"  NS <sub>1</sub> : ???

**C<sub>11</sub>**: valores de entrada correspondientes a un triángulo equilátero

**C<sub>12</sub>**: valores de entrada correspondientes a un triángulo isósceles

**C<sub>13</sub>**: valores de entrada correspondientes a un triángulo escaleno

**C<sub>14</sub>**: valores de entrada que se corresponden con valores válidos pero que no forman un triángulo

Las clases **C<sub>11</sub>, C<sub>12</sub>, y C<sub>13</sub>** incluyen, además, la condición es-triángulo

Qué ocurre si la entrada es NC<sub>i</sub>?

# EJEMPLO 3: TABLA DE CASOS DE PRUEBA

- La tabla resultante de casos de prueba puede ser ésta:

Clases	Datos Entrada	Resultado Esperado
C11-S1	a=11, b=11, c=11	"Equilátero"
C12-S2	a=7, b=7, c=6	"Isósceles"
C13-S3	a=10, b=3, c=9	"Escaleno"
C14-S4	a=8, b=2, c=4	"No es triángulo"
NC1-S5	a=30, b=15, c=6	???
NC2-S5	a=10, b=100, c=10	???
NC3-S5	a=7, b=14, c=21	???
NC4-S5	a=-5, b=10, c=11	???
NC5-S5	a=12, b=-10 c=10	???
NC6-S5	a=8, b=5, c=-1	???

1 Salida

3 Entradas



No debes asumir un resultado esperado que no esté indicado en la especificación

# ALGUNOS CONSEJOS...



- Etiqueta las particiones de una misma E/S con la misma letra.
  - Por ejemplo, si las entradas son p1 y p2, las particiones podrían etiquetarse como A1, A2,... NA1, NA2,..., B1, B2,... NB1, NB2,... para las clases válidas e inválidas del parámetro p1 y p2 respectivamente
- No olvides tener en cuenta las precondiciones de entrada/salida al realizar las particiones
- Si las E/S son objetos, tendrás que considerar cada atributo del objeto como un parámetro diferente.
  - P.ej. supón que una entrada es el objeto coordenadas (c), el cual tiene como atributos, valores de "x" e "y". Tendrás que hacer las particiones tanto para "c.x" como para "c.y"
- Los objetos en java siempre son referenciados por las variables. Por lo tanto tendremos que considerar el valor NULL como partición no válida al usar objetos
- Las E/S NO son ÚNICAMENTE parámetros de la unidad a probar.
  - P.ej. en el método realizaReserva() que hemos visto en prácticas, el resultado de invocar a la unidad reserva(socio.isbn) es una entrada para el método realizarReserva() que debemos incluir en la tabla de casos de prueba.

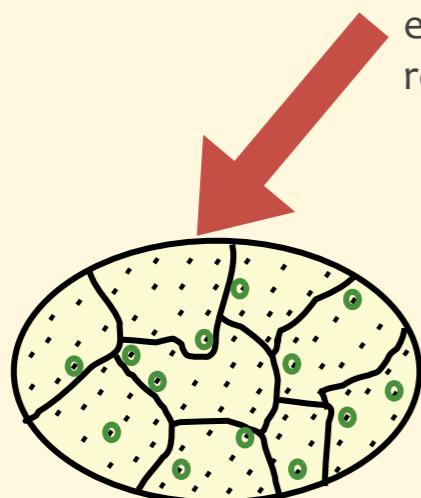
# Y AHORA VAMOS AL LABORATORIO...

Usaremos el método de particiones equivalentes

Identificaremos casos de prueba utilizando caja negra

## ESPECIFICACIÓN

(unidad)



Método de  
particiones  
equivalentes

A nivel de unidad

Supongamos que queremos realizar pruebas sobre un **método** que calcula el nuevo importe de la renovación anual de una póliza de seguros. Si el asegurado es mayor de 25 años, y no tiene ningún parte de reclamación registrado en el último año, entonces se le incrementa en 25 euros el valor de la póliza, si tiene una reclamación, entonces se le incrementa en 50 euros, si tiene entre 2 y 4 reclamaciones, la actualización será de 200 euros y se le envía una carta al asegurado. Si el asegurado tiene 25 años o menos y ninguna reclamación cursada, la póliza se actualiza en 50 euros más. Si tiene una reclamación, se incrementa en 100 euros y se le envía una carta. Entre 2 y 4 reclamaciones, el incremento será de 400 euros y también se le envía una carta. Independientemente de la edad, si un asegurado tiene más de cinco reclamaciones se le cancelará la póliza

Entrada 1: Particiones válidas: A1, A2, A3  
Particiones inválidas : NA1, NA2

...  
Entrada k: Particiones válidas: K1, K2, K3  
Particiones inválidas: NK1

Salida 1: Particiones válidas: S1, S2, S3  
Particiones inválidas: NS1, NS2

...  
Salida p: Particiones válidas: P1, P2  
Particiones inválidas: NP1, NP2

## Tabla de casos de prueba

Particiones	Identificador	Datos Entrada	Resultado Esperado
A1- B1- ... - K1	C1	d1=... d2=... ...	r1= ... r2= ...
...	...		
AX- BY- ... - NKZ	CM	d1=... d2=... ...	r1= .. r2= ...

La utilizaremos en la siguiente práctica!!!...

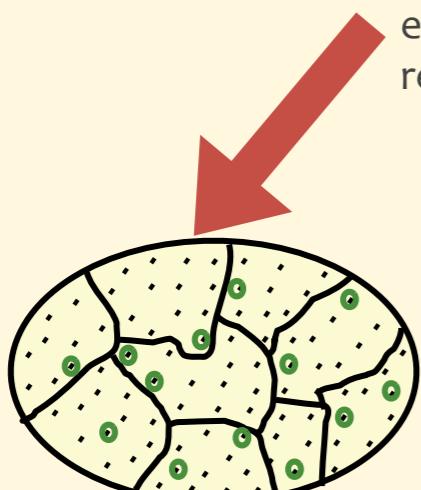
# P Y AHORA VAMOS AL LABORATORIO...

P Identificaremos casos de prueba utilizando métodos de caja negra

Supongamos que queremos realizar pruebas sobre un **método/sistema** que calcula el nuevo importe de la renovación anual de una póliza de seguros. Si el asegurado es mayor de 25 años, y no tiene ningún parte de reclamación registrado en el último año, entonces se le incrementa en 25 euros el valor de la póliza, si tiene una reclamación, entonces se le incrementa en 50 euros, si tiene entre 2 y 4 reclamaciones, la actualización será de 200 euros y se le envía una carta al asegurado. Si el asegurado tiene 25 años o menos y ninguna reclamación cursada, la póliza se actualiza en 50 euros más. Si tiene una reclamación, se incrementa en 100 euros y se le envía una carta. Entre 2 y 4 reclamaciones, el incremento será de 400 euros y también se le envía una carta. Independientemente de la edad, si un asegurado tiene más de cinco reclamaciones se le cancelará la póliza

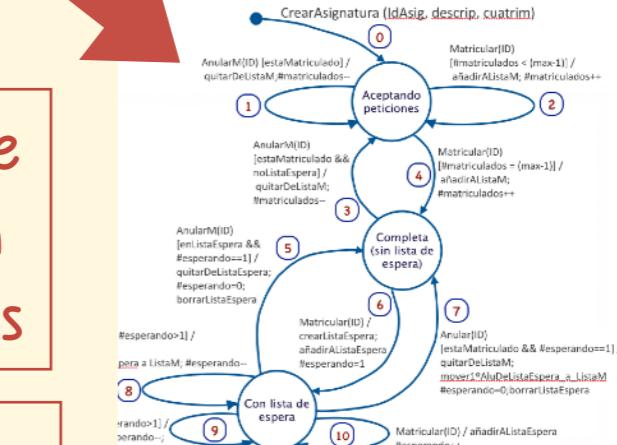
## ESPECIFICACIÓN

(unidad o sistema)



Método de  
particiones  
equivalentes

A nivel de unidad



Método de  
transición  
de estados

A nivel de sistema

Tabla de casos de prueba

Identificador CP	Datos Entrada	Resultado Esperado
C1	d1=... d2=... ... dk=...	r1
...		
CM	d1=... d2=... ... dk=...	rM

La utilizaremos en la siguiente práctica!!!...

# REFERENCIAS



- A practitioner's guide to software test design. Lee Copeland.  
Artech House Publishers. 2007
  - Capítulo 3: Equivalence Class Testing
  - Capítulo 7: State-Transition Testing
- Pragmatic software testing. Rex Black. Wiley. 2007
  - Capítulo 11: Equivalence Classes Exercise
  - Capítulo 14: State-Transition Diagrams
  - Capítulo 15: State-Transition diagram Exercise