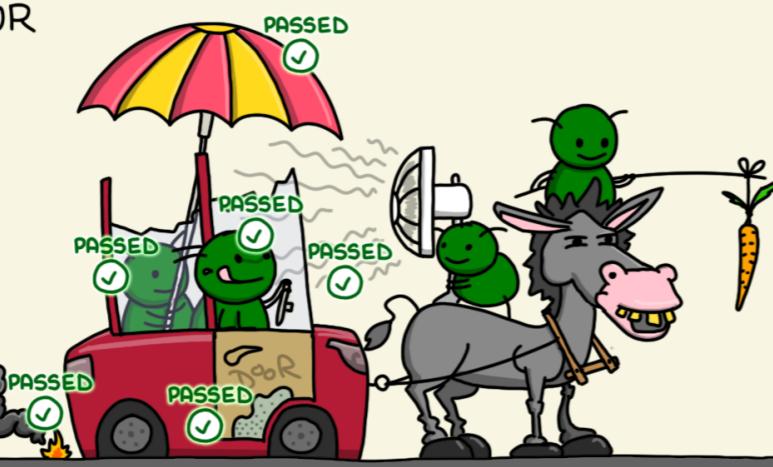


Sesión S04: Diseño de pruebas: **caja negra**

UNIT TESTS

MONKEYUSER.COM

SO YOU ASSUMED
THAT THIS IS THE
CORRECT BEHAVIOR



Las especificaciones SIEMPRE definen los comportamientos esperados. NUNCA asumas que el comportamiento implementado es el correcto!!

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, dado un objetivo, el máximo número posible de defectos en el código, con el mínimo número posible de "filas" (efectividad y eficiencia)

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

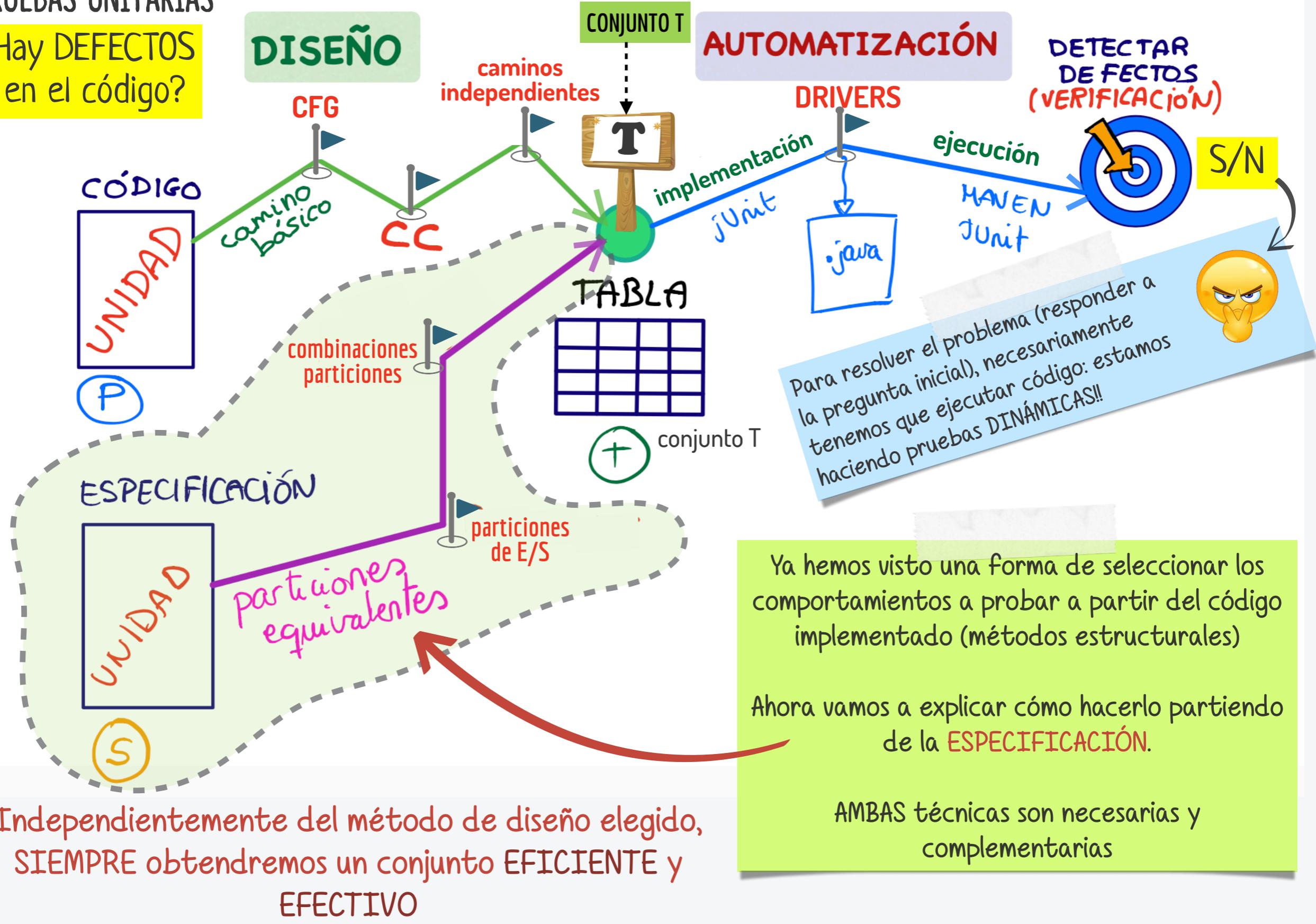
Vamos al laboratorio...

DISEÑO DE CASOS DE PRUEBA

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

PRUEBAS UNITARIAS

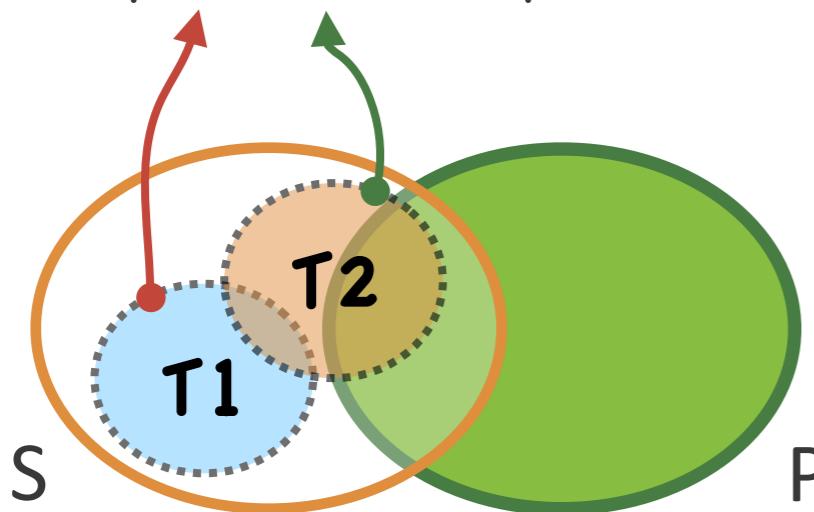
Hay DEFECTOS en el código?



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

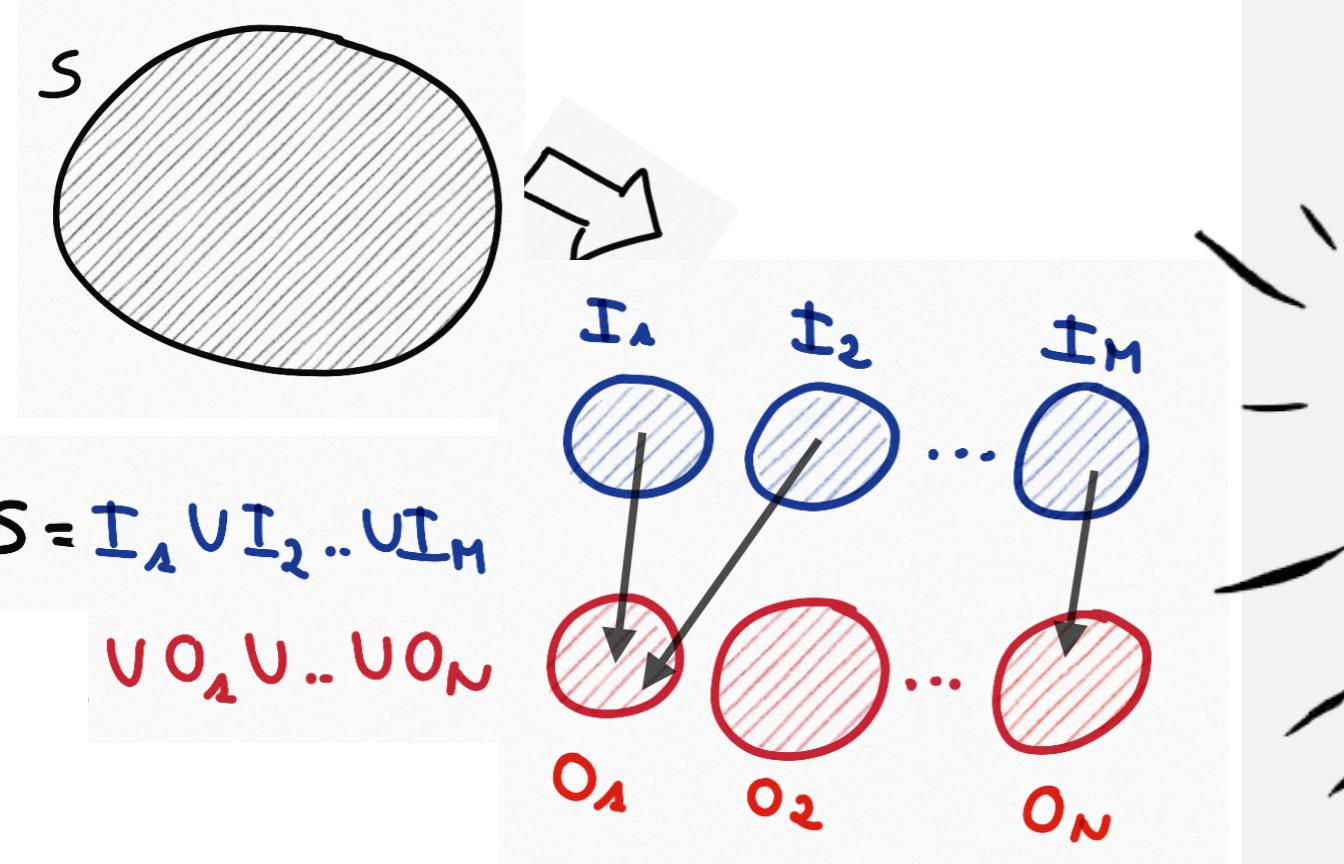
- Se trata de un proceso **SISTEMÁTICO** que identifica, a partir de la **ESPECIFICACIÓN** disponible, un conjunto de **CLASES** de equivalencia para **cada** una de las **entradas** y **salidas** del "elemento" (unidad, componente, sistema) a probar
- Cada clase de equivalencia (o partición) de entrada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

P

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.

MÉTODO DE DISEÑO: PARTICIONES EQUIVALENTES

Sesión 4: Diseño de caja negra



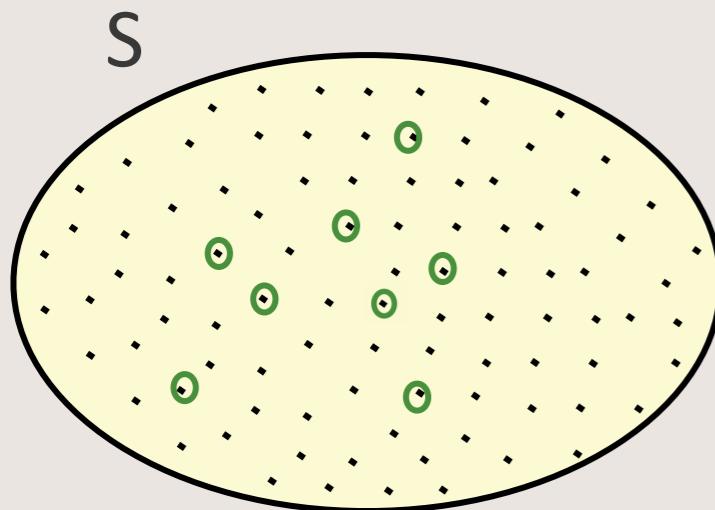
- Cada caso de prueba usará un **subconjunto** de particiones
- **NO** se trata de probar **TODAS** las combinaciones posibles, sino de garantizar que **TODAS** las particiones de entrada (y de salida) se prueban **AL MENOS UNA VEZ**

SISTEMATICIDAD Y PARTICIONAMIENTOS

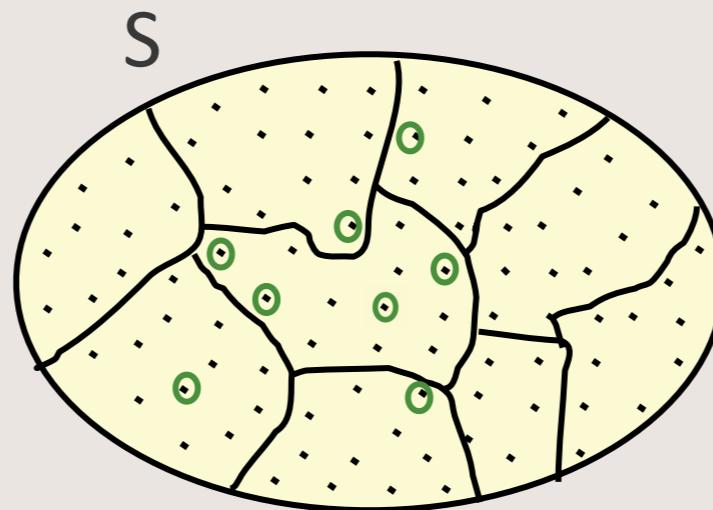
detectar el máximo nº posible de errores

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

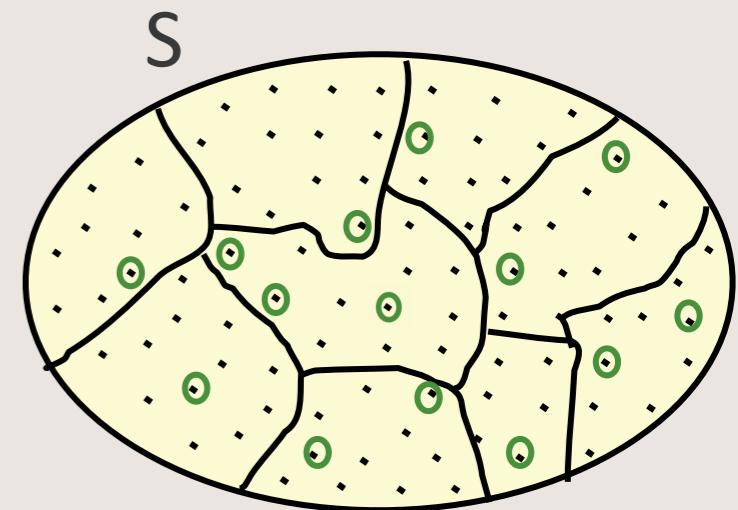
- Para conseguir un conjunto de pruebas **EFFECTIVO** y **EFICIENTE**, tenemos que ser **SISTEMÁTICOS** a la hora de determinar las particiones de entrada/ salida
 - Las particiones representan conjuntos de posibles comportamientos del sistema
 - Se deben elegir muestras significativas de CADA partición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



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



Particiones. Se eligen muestras de cada partición



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

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

EL TIPO de una variable determina el conjunto de valores posibles para esa variable

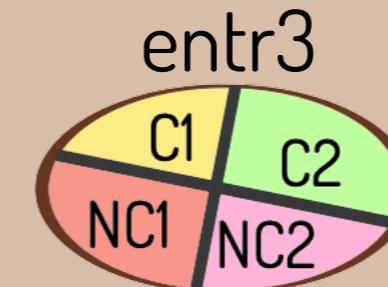
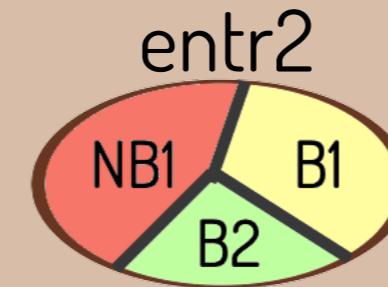
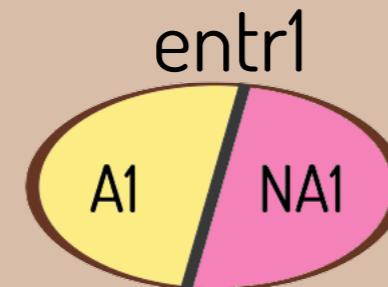
- Las "variables" de entrada/salida **no necesariamente** se corresponden con "parámetros" de entrada/salida de la unidad a probar
 - P.ej. El método `validar_pin()` comprueba si un pin (obtenido invocando a otra unidad) es válido o no. Si es válido, el método devuelve también el pin obtenido...
 - * Supongamos que el método a probar es:
 - **boolean validar_pin(Pin pinValido)**
 - * Las "variables" de entrada/salida que debemos considerar son:
 - Entradas: (1) 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, (2) valor booleano que indica si el pin es válido
 - Salida: booleano + objeto pin válido
- Las particiones deben ser DISJUNTAS (las particiones No comparten elementos).
 - P.ej. Dada una entrada "a" que representa un entero, las particiones: (1) $a \geq 10$, y (2) $a \leq 10$ NO son disjuntas puesto que el valor $a=10$ pertenece a dos particiones diferentes
- 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)



Cada Entrada (Salida) se almacena en una variable, por lo tanto tendremos que tener en cuenta el DOMINIO (valores posibles) de dicha variable para hacer las particiones!!

ETIQUETAREMOS LAS PARTICIONES

Una partición puede ser VÁLIDA o INVÁLIDA (prefijo N)



Particionamos CADA entrada y salida

```
salida SUT(T1 entr1, T2 entr2, T3 entr3) {
    T4 salida;
    ...
    return salida;
}
```

Las particiones son DISJUNTAS

La unión de particiones de una variable = dominio de valores de dicha variable

- El **TIPO** de una variable determina cuales son los posibles valores que puede tomar esa variable
- Para particionar cada entrada/salida aplicamos las **condiciones** sobre dicha variable indicadas en la **especificación**
- Cada partición sobre una variable se considerará **VÁLIDA** o **INVALIDA** (según unas heurísticas)
- En CADA caso de prueba usamos una partición para CADA entrada/salida
 - Para el ejemplo anterior, cada caso de prueba usa 4 particiones porque tenemos 3 entradas y 1 salida

P IDENTIFICACIÓN DE LAS CLASES DE EQUIVALENCIA



PASO 1: Identificar las clases de equivalencia (particiones) para CADA entrada/salida (E/S), usando 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

PASO 2: Identificar los casos de prueba de la siguiente forma:

Debemos asignar un IDENTIFICADOR ÚNICO para cada partición



El orden
de los
pasos
importa!!

- 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 CARACTERES

3 Entradas + 1 Salida

ESPECIFICACIÓN: Método en el que, dados como entradas: un carácter X cuaiquiera introducido por el usuario, un número N entre 5 y 10, y el valor “rojo” o “azul” (elegidos de una lista desplegable), devuelve 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.

Prototipo del método: `String impresor(char x, int n, String color)`

- **Entrada 1 (C):** carácter (car)

- Clase válida: **C1** : car puede ser cualquier carácter

- **Entrada 2 (U):** número (n)

- Clase válida: **U1**: $(5 \leq n \leq 10)$

- Clases inválidas: **NU1**: $(n < 5)$

- NU2**: $(n > 10)$

car, n, c son las variables que representan las entradas **1, 2 y 3** respectivamente y que tenemos que particionar

- **Entrada 3 (L):** color (c)

- Clases válidas: **L1**: (c=“rojo”)

- L2**: (c = “azul”)

cad es la variable de salida

- **Salida (S):** cadena de n caracteres (n es el valor de entrada 2) (cad)

- Clase válida: **S1**: cad está formada por n caracteres de color rojo

- Clase válida: **S2**: cad está formada por (n-1) caracteres de color azul

- Clase inválida: **NS1**: “ERROR: repite entrada”

C, U, L, S son las etiquetas de las particiones válidas de E/S

Particiones	Datos Entrada			Resultado Esperado
	car	n	col	
C1-U1-L1-S1	‘c’	7	“rojo”	“cccccc”
C1-U1-L2-S2	‘x’	6	“azul”	“xxxxxx”
C1-NU1-L2-NS1	‘c’	3	“azul”	“ERROR: repite entrada”
C1-NU2-L2-NS1	‘j’	13	“azul”	“ERROR: repite entrada”

EJEMPLO 2: VALIDAR FECHA

2 Entradas + 1 Salida

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 2025 (no bisiesto)

○ 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

(Paso 1)

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

- Entradas 1+2 (DM):** agrupamos las entradas ($d + m$) que representan el día y el mes, respectivamente

- Clases válidas:

DM1: $d = \{1..28\} \wedge m = \{1..12\}$

DM2: $d = \{29,30\} \wedge m = \{1,3,..,12\}$

DM3: $d = \{31\} \wedge m = \{1,3,5,7,8,10,12\}$

- Clases inválidas:

NDM1: $d > 31 \wedge m = \{1..12\}$

NDM2: $d < 1 \wedge m = \{1..12\}$

NDM3: $d = \{29,30\} \wedge m = \{2\}$

NDM4: $d = 31 \wedge m = \{2,4,6,9,11\}$

NDM5: $m > 12 \wedge d = \{1..31\}$

NDM6: $m < 1 \wedge d = \{1..31\}$

- Salida 1 (S):** valor booleano (s)

- Clases válidas:

S1: s = true

- Clases inválidas:

NS1: s = false



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() (PASOS 2 Y 3)

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

(Paso 2)

Primero obtenemos las combinaciones, y luego rellenamos la tabla con valores concretos

Si las particiones válidas las etiquetamos como DM1, DM2 ...



Las particiones inválidas las etiquetamos como NDM1, NDM2 ...

Particiones	d	m	s
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!!!!

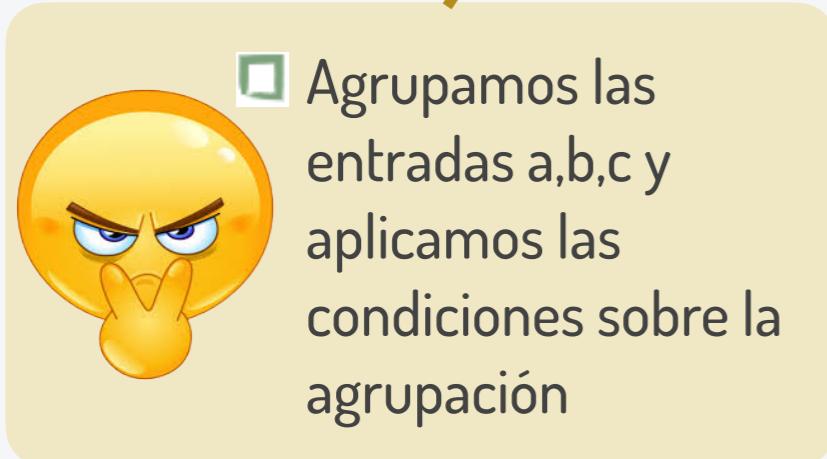


EJEMPLO 3: EL PROBLEMA DEL TRIÁNGULO

P **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 los tres lados son desiguales
- * "No es un triángulo", si $a \geq b+c$, ó $b \geq a+c$, ó $c \geq a+b$

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



Entradas 1+2+3 (C) : valores a, b, c

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

NC1, NC2, ... NC6 incluyen valores válidos (> 0 y ≤ 20) para las otras dos variables.

P

EJEMPLO 3: PARTICIONES

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 ₁₁ : a=b=c	NC ₁ = a > 20	S ₁ : "Equilátero"
C ₁₂ : (a=b \wedge a <> c) \vee (a=c \wedge a <> b) \vee (b=c \wedge b <> a)	NC ₂ = b > 20	S ₂ : "Isósceles"
C ₁₃ : (a <> b) \wedge (a <> c) \wedge (b <> c)	NC ₃ = c > 20	S ₃ : "Escaleno"
C ₁₄ : (a ≥ b+c) \vee (b ≥ a+c) \vee (c ≥ a+b)	NC ₄ = a ≤ 0	S ₄ : "No es triángulo"
	NC ₅ = b ≤ 0	NS ₁ : ???
	NC ₆ = c ≤ 0	

C₁₁: valores de entrada correspondientes a un triángulo equilátero

C₁₂: valores de entrada correspondientes a un triángulo isósceles

C₁₃: valores de entrada correspondientes a un triángulo escaleno

C₁₄: valores de entrada que no forman un triángulo

Las clases **C₁₁**, **C₁₂**, y **C₁₃** incluyen, además, la condición es-triángulo

Las clases **C₁₁**, **C₁₂**, **C₁₃** y **C₁₄** incluyen, además, la condición cd valores válidos (C1)

Qué ocurre si la entrada es NC_i?

EJEMPLO 3: TABLA DE CASOS DE PRUEBA

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

1 Salida

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-NS1	a=30, b=15, c=6	???
NC2-NS1	a=10, b=100, c=10	???
NC3-NS1	a=7, b=14, c=21	???
NC4-NS1	a=-5, b=10, c=11	???
NC5-NS1	a=12, b=-10 c=10	???
NC6-NS1	a=8, b=5, c=-1	???

3 Entradas



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

ALGUNOS CONSEJOS...



P

O Etiqueta las particiones de una misma E/S con la misma letra.



- Por ejemplo, si etiquetamos una partición con el valor A, las particiones válidas se llamarán A1, A2,... y las inválidas NA1, NA2,...
- No olvides tener en cuenta las precondiciones de entrada/salida al realizar las particiones
 - P.ej. los valores de entrada se eligen de una lista desplegable (por lo tanto no debemos considerar la partición formada por los elementos que no pertenecen al conjunto)

O Si las E/S son objetos, tendrás que considerar cada atributo del objeto como un parámetro diferente.

- P.ej. supón que una entrada (c) es un objeto de tipo Coordenadas, el cual tiene como atributos, valores de "x" e "y". Tendrás que hacer las particiones tanto para "c.x" como para "c.y"

O Los objetos en java siempre se almacenan siempre por referencia. Por lo tanto tendremos que considerar el valor NULL como partición no válida al usar objetos

O Las E/S NO son ÚNICAMENTE parámetros de la unidad a probar.

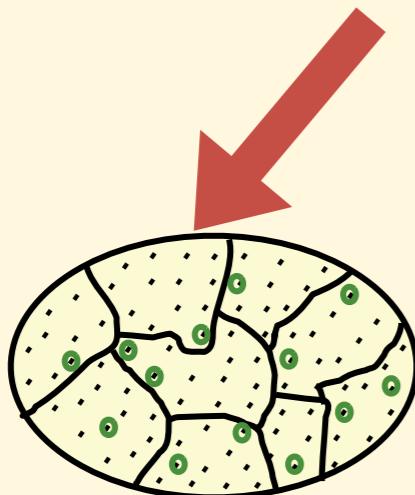
- P.ej. en el método realizaReserva() que hemos visto en prácticas, el resultado de invocar a la unidad reserva(socio.isbn) es una entrada para el método realizarReserva() que debemos incluir en la tabla de casos de prueba.

Y AHORA VAMOS AL LABORATORIO...

Usaremos el método de particiones equivalentes

ESPECIFICACIÓN

(unidad)



Método de
particiones
equivalentes

A nivel de unidad

En una aplicación de un negocio de alquiler de coches necesitamos una unidad denominada `importe_alquiler_coche()`. Dicha unidad calcula el importe del alquiler de un determinado tipo de coche durante un cierto número de días, a partir de una fecha concreta, y devuelve el importe de dicho alquiler. Si no es posible realizar los cálculos devuelve una excepción de tipo `ReservaException`. Asumimos que la fecha de inicio ha sido validad en otra unidad. Nos indican que si la fecha de inicio proporcionada no es posterior a la actual, entonces se lanzará la excepción `ReservaException` con el mensaje "Fecha no correcta". Si el tipo de coche no está disponible durante los días requeridos, o se intenta hacer una reserva de más de 30 días, entonces se lanzará la excepción `ReservaException` con el mensaje "Reserva no posible"

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

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

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

...
Salida N (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 - NBY - ... - NKZ	CM	d1=... d2=... ...	r1 = .. r2 = ...

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
- Pragmatic software testing. Rex Black. Wiley. 2007
 - Capítulo 11: Equivalence Classes Exercise

VIDEOS INTERESANTES



[Black and White Box Testing Introduction - Georgia Tech - Software Development Process](#)



[Black Box testing Overview - Georgia Tech - Software Development Process](#)



[Why Not Random Testing? - Georgia Tech - Software Development Process](#)



[Partition Testing Example - Georgia Tech - Software Development Process](#)