
Planificación y Pruebas de Sistemas Software

Curso 2015-16

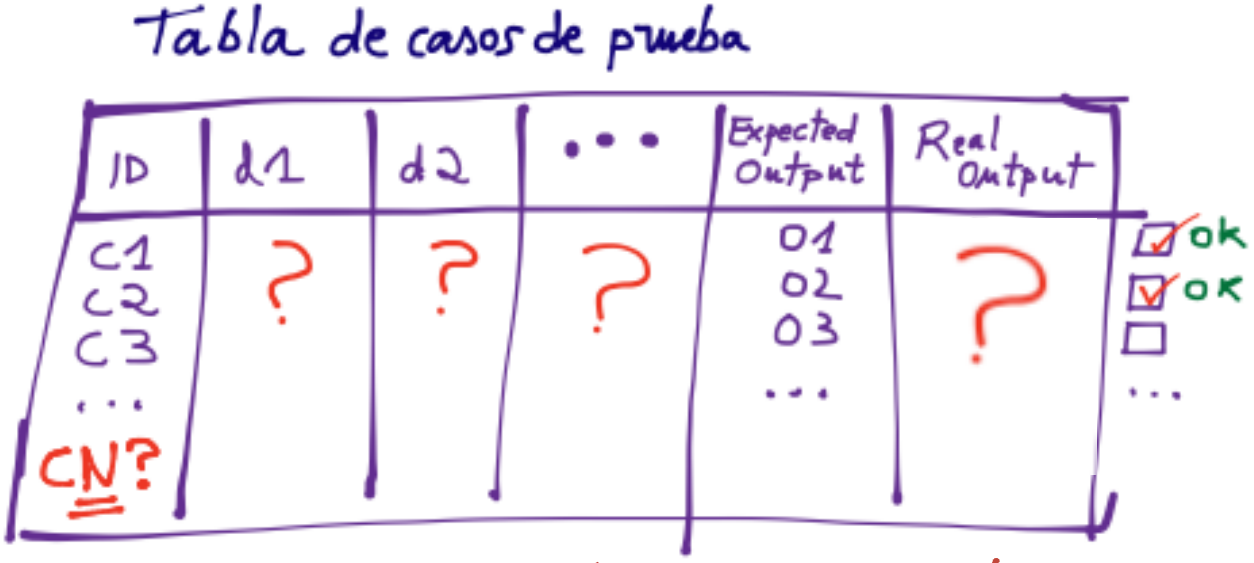
# Sesión 4: Diseño de pruebas de caja negra

Diseño de casos de pruebas funcionales
Método de particiones equivalentes
Método de transición de estados

María Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

# Diseño de casos de prueba

- Vamos a ver otros métodos para poder determinar un conjunto de casos de prueba eficiente y efectivo
  - Recuerda que un buen diseño de casos de prueba es fundamental para conseguir nuestro objetivo: detectar el mayor número posible de defectos en el software, para poder satisfacer las necesidades del cliente y contribuir al éxito del proyecto



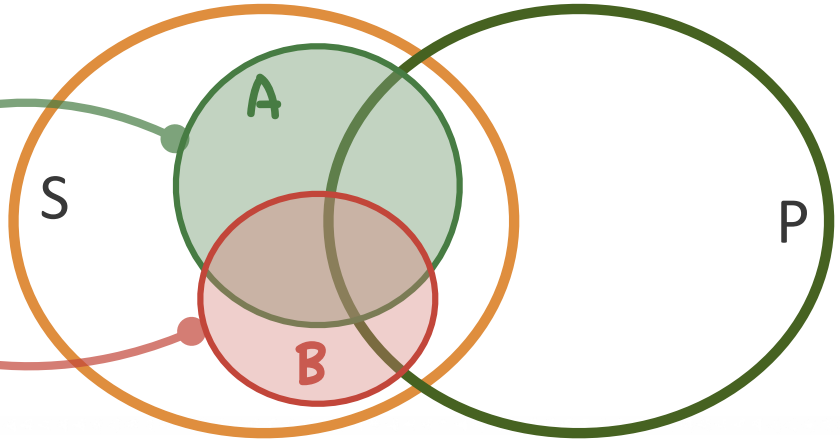

podemos obtener un conjunto de casos de prueba eficientes y efectivos utilizando múltiples MÉTODOS de DISEÑO de casos de prueba!!!!!

# Formas de identificar los casos de prueba

DISEÑAR

- Fundamentalmente hay dos formas de proceder:
  - ***Functional testing***: aproximación basada en la ESPECIFICACIÓN
    - 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
  - ***Structural testing***: aproximación basada en la IMPLEMENTACIÓN
    - Utilizamos EL CÓDIGO para determinar el conjunto de casos de prueba. Esta aproximación también se conoce con el nombre de "caja blanca"
    - Es esencial conocer conceptos de teoría de grafos para entender bien esta aproximación

Métodos de diseño de caja NEGRA




# Métodos funcionales. Observaciones

- Los métodos funcionales:
  - Aplican un proceso analítico que descompone la especificación de un programa en diferentes clases de comportamientos
  - A continuación seleccionan ciertas combinaciones de comportamientos según algún criterio
  - Obtienen un conjunto de casos de prueba que ejercitan dichos comportamientos
- Dependiendo del método utilizado, obtendremos conjuntos DIFERENTES de casos de prueba:
  - Pero el conjunto obtenido será EFECTIVO y EFICIENTE!!!!
- Las técnicas o métodos funcionales pueden aplicarse a cualquier nivel de pruebas (unidad, integración, sistema, aceptación)
- Los métodos funcionales no pueden detectar todos los defectos en el programa (faults, bugs)
  - Aunque seleccionemos todas las posibles entradas, no podremos detectar todos los defectos si dejamos de probar alguna combinación de éstas (las pruebas exhaustivas son imposibles)

# Métodos de diseño de caja negra

- Existen múltiples métodos de diseño de pruebas de caja negra:
  - Método de particiones equivalentes
  - Método de análisis de valores límite
  - Método de tablas de decisión
  - Método de grafos causa-efecto
  - Método de diagramas de transición de estados
  - Método de pruebas basado en casos de uso
  - Método de pruebas basado en requerimientos
  - Método de pruebas basado en escenarios

Pruebas unitarias

Pruebas de sistema

Pruebas de aceptación

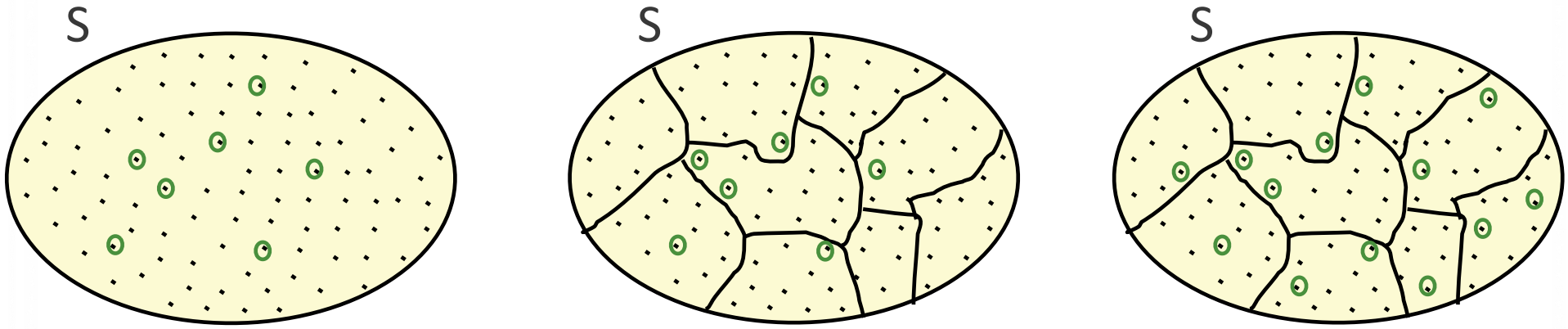
En todos ellos, la **identificación de dominios de entradas y salidas** contribuye a **particionar** las clases de comportamientos


# Método de Particiones equivalentes

- El método de diseño de casos de prueba de particiones equivalentes es un proceso SISTEMÁTICO que identifica, a partir de la especificación disponible, un conjunto de CLASES de equivalencia de entrada y de salida para el "elemento" (unidad, componente, sistema) a probar
  - Cada clase de equivalencia (o partición) de entrada representa un subconjunto del total de datos posibles de entrada. Los elementos de una misma partición de entrada se caracterizan por tener su "imagen" en la misma partición de salida
- El OBJETIVO es MINIMIZAR el número de casos de prueba requeridos para cubrir TODAS las particiones al menos una vez
  - Elegiremos UN caso de prueba para cada partición
  - NO se trata de probar TODAS las combinaciones de entradas, sino de garantizar que TODAS las particiones de entrada (y de salida) se prueban AL MENOS UNA VEZ (tenemos que cubrir TODAS las particiones)

# Sistematicidad y particionamientos

- Para conseguir un conjunto de pruebas eficiente y efectivo, 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

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

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

# ¿Como identificamos una partición?

- 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)
- 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 ≤ 20
      - (2) a >20
      - (3) b > 20
      - (4) c > 20
      - ...

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

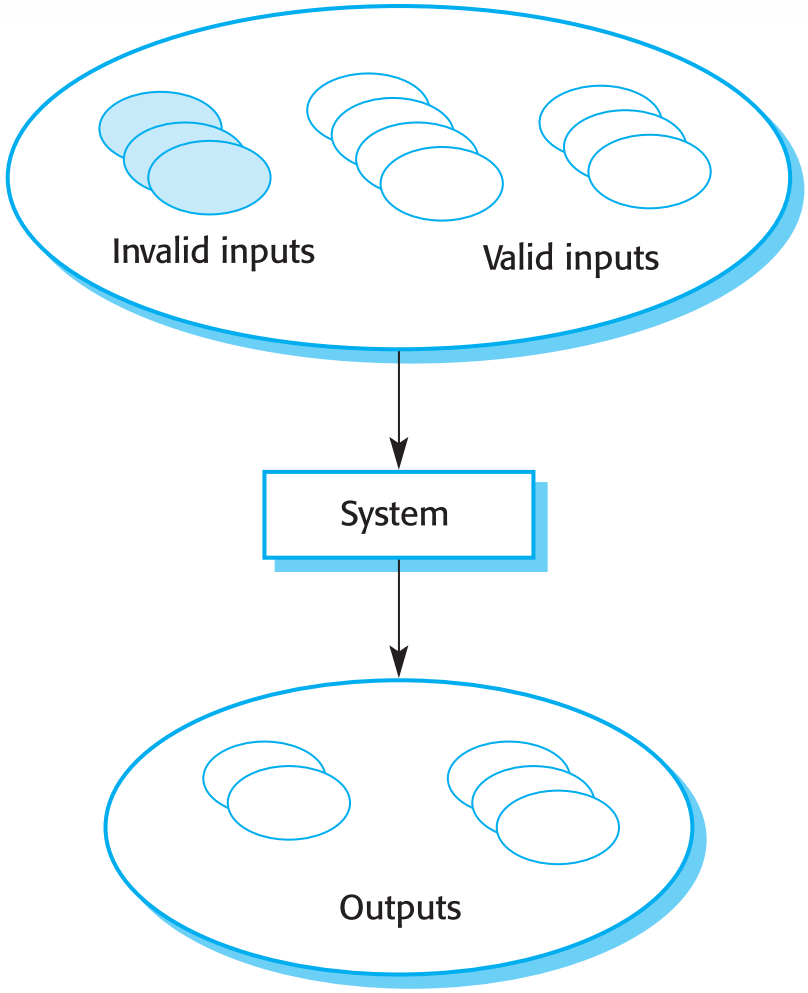
la condición de entrada sólo se aplica a una variable


# 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. Queremos probar un método que añade los datos referentes a un alumno en una base de datos. Si se intenta añadir (dar de alta) a un alumno que ya existe el método debe proporcionar un mensaje de error...
    - Supongamos que el método a probar es:
      - `String alta_alumno(Alumno alu)`
    - Las "variables" de entrada que debemos considerar son:
      - Alumno
      - Estado de la tabla alumno en la BD antes de ejecutar el método
- 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

- 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.




# Identificación de las clases de equivalencia

- **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 ϵ al conjunto; Clase inválida: x toma cualquier valor que no ϵ 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:
  - Asignar un identificador único para cada partición
  - 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
  - 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 se prueban múltiples clases inválidas en un mismo caso de prueba, puede que alguna de estas clases no se ejecuten nunca, ya que alguna de las clases no válidas puede "enmascarar" a alguna otra, o incluso terminar con la ejecución del caso de prueba
  - El resultado será una TABLA con tantas FILAS como CASOS de PRUEBA hayamos obtenido

# Ejemplo 1: Impresión de caracteres

**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 (carácter X): puede ser cualquier carácter
  - Clase válida: V1
- Entrada 2 (número N): un valor comprendido entre 5 y 10
  - Clase válida: V2: valores entre 5 y 10 (5 ≤ N ≤ 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"

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

| Clases | Datos Entrada | Resultado Esperado | Resultado Real |
|---|---|---|---|
| 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

- Supongamos que tenemos el método valida_fecha() que tiene como parámetros de entrada las variables de tipo entero: dia y mes, de forma que dados ambos valores, devuelva cierto o falso, en función de que sea una fecha válida. Supongamos que el año es 2016. 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 como entrada:
    - (dia, mes)
  - 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


# Ejemplo 2: Validar Fecha (particiones)

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

(Paso 1)

| Particiones | |
|---|---|
| dia + mes | salida |
| DM1: d ={1..29} ∧ m= {1..12} | S1: true |
| DM2: d ={30} ∧ m= {1,3,..,12} | NS1: false |
| DM3: d ={31} ∧ m= {1,3,5,7,8,10,12} | |
| NDM1: d > 31 ∧ m= {1..12} | |
| NDM2: m >12 ∧ d= {1..31} | |
| NDM3: d < 1 ∧ m= {1..12} | |
| NDM4: m < 1 ∧ d= {1..31} | |
| NDM5: d=30 ∧ m= {2} | |
| NDM6: d=31 ∧ m= {2,4,6,9,11} | |


# 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 | 29 | 16 | false |
| NDM3-NS1 | -3 | 6 | false |
| NDM4-NS1 | 29 | -3 | false |
| NDM5-NS1 | 30 | 2 | false |
| NDM6-NS1 | 31 | 4 | false |

# Ejemplo 3: El problema del triángulo

- **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 ≥b+c, b ≥a+c, ó c ≥a+b
- Paso 1. Inicialmente podemos definir las siguientes particiones de entrada:

| Entrada: a,b,c | |
|---|---|
| C1: a,b,c >0 ∧ a,b,c ≤ 20 | NC1 = a >20 |
| | NC2 = b >20 |
| | NC3 = c >20 |
| | NC4 = a ≤ 0 |
| | NC5 = b ≤ 0 |
| | NC6 = c ≤ 0 |

# Ejemplo 3: Particiones

- 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 ∧ b <a+c ∧ c <a+b
- También particionamos las salidas:

| Entrada: a,b,c | | Salida |
|---|---|---|
| C11: a=b=c | NC1 = a >20 | S1: "Equilátero" |
| C12: (a=b ∧ a<>c) ∨ (a=c ∧ a <>b) ∨ (b=c ∧ b<>a) | NC2 = b >20 | S2: "Isósceles" |
| C13: (a<>b) ∧ (a <>c) ∧ (b<>c) | NC3 = c >20 | S3: "Escaleno" |
| C14: (a ≥b+c) ∨ (b ≥a+c) ∨ (c ≥a+b) | NC4 = a ≤ 0 | S4: "No es triángulo" |
| | NC5 = b ≤ 0 | S5: ??? |
| | NC6 = 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 se corresponden con valores válidos pero que no forman un triángulo
Las clases **C11**, **C12**, y **C13** incluyen, además, la condición es-triángulo


# Ejemplo 3: Tabla de casos de prueba

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

| Clases | Datos Entrada | Resultado Esperado | Resultado Real |
|---|---|---|---|
| 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=99 | ??? | |
| 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 | ??? | |


# Algunos consejos...

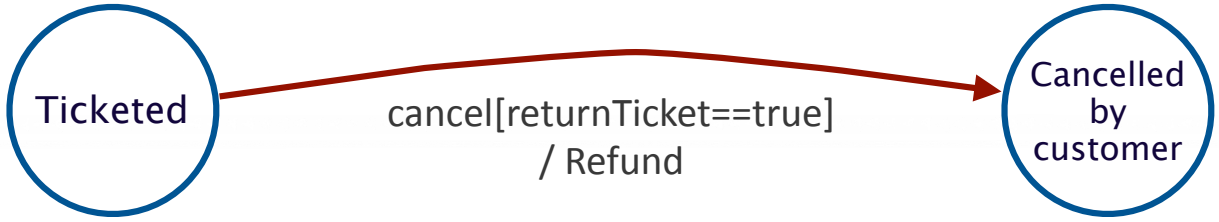
- Etiqueta las particiones de una misma E/S con la misma letra. Por ejemplo, si las entradas son p1 y p2, las particiones podrían etiquetarse como A1, A2,... NA1, NA2,..., B1, B2,... NB1, NB1,... para las clases válidas e inválidas del parámetro p1 y p2 respectivamente
- No olvides tener en cuenta las 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"
- Las E/S NO son ÚNICAMENTE parámetros de la unidad a probar. P.ej. supongamos un método, que dado como entrada un identificador de usuario, busca al usuario en una BD e inserta el nuevo identificador en la BD si éste no existe. En este caso las entradas son: el identificador, y el estado de la base de datos ANTES de la llamada al método. La salida es el estado de la base de datos DESPUÉS de la llamada al método


# Método de diagrama de transición de estados

- Los diagramas de transición de estados (STD: State-Transition Diagram) tienen en cuenta ocurrencia de eventos y su procesamiento por parte del sistema, así como la respuesta del sistema ante dichos eventos
- En este caso las reglas de procesamiento pierden "protagonismo" en favor de los cambios en el estado del sistema (valores de conjuntos de variables). Cuando un sistema debe recordar algo que ocurrió con anterioridad, o establecer que conjunto de acciones son o no posibles en un instante dado, entonces los diagramas de transición de estado permiten registrar dicha información
- Por lo tanto, el objetivo es detectar errores en los "cambios de estado" de un **sistema** durante su ejecución
- El sistema se modela como una máquina de estados. Un modelo basado en estados está formado por los siguientes componentes:
  - **Estados:** representan el "impacto acumulado" de entradas al sistema pertenecientes al pasado. Son condiciones en las que el sistema está esperando la ocurrencia de algún evento
  - **Transiciones:** representan cambios el estado en respuesta a determinados eventos
  - **Eventos:** son entradas del sistema que provocan posibles cambios de estado
  - **Acciones:** son las operaciones iniciadas como consecuencia de la ocurrencia de un evento

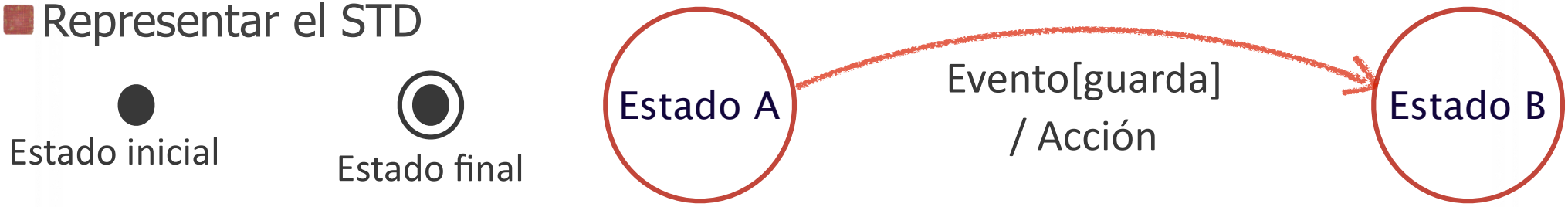
# Transiciones, Eventos y Acciones

- Cada diagrama de transición de estados representa UNA entidad específica, por ejemplo, una RESERVA de billete de avión
  - El diagrama debe describir los estados que afectan a la reserva, las transiciones de la reserva de un estado a otro, y las acciones que se llevan a cabo durante la reserva. Un error común es incluir diferentes entidades en el mismo diagrama (por ejemplo, reservas y pasajeros, con eventos y acciones de cada uno de ellos)
- Los eventos provocan (o pueden provocar) cambios de estado (transiciones), y generalmente son "externos" al sistema, y son "capturados" a través de su interfaz. En otras ocasiones pueden ser generados dentro del sistema, como por ejemplo "la finalización de un timer". Los eventos se considera que son "instantáneos"
  - Cuando ocurre un evento, el sistema puede cambiar de estado o permanecer en el mismo estado y/o ejecutar una acción.
  - Los eventos pueden tener parámetros asociados (guardas) que representan condiciones que permiten discriminar entre posibles estados resultantes.
- Una acción se representa por un comando precedido de "**/**". A menudo las acciones generan salidas del sistema. Las acciones tienen lugar en las transiciones entre estados. Los estado en sí mismos son pasivos



# Pasos a seguir para diseñar las pruebas

- A partir de la especificación, determinar los conjuntos de:
  - estados: cada estado viene dado por un conjunto de valores de variables del sistema (se representan en los nodos del STD)
  - eventos: entradas al sistema que pueden provocar cambios de estado (se indican en las aristas del STD). Los eventos pueden tener parámetros asociados (o guardas) que representan condiciones que permiten discriminar entre diferentes posibles estados resultantes
  - acciones: operaciones iniciadas por la ocurrencia de un evento (se indican en las aristas del STD)
  - transiciones: cambios entre estados (aristas del STD).
- Representar el STD




- Seleccionar un conjunto de caminos según un determinado criterio
  - Cubrir todas las transiciones del diagrama (grafo)


# Cómo crear *tests* a partir de un STD

- Obviamente, probar todas las combinaciones posibles de transiciones de estados puede resultar impracticable, por lo que, una vez creado el diagrama, lo utilizaremos para crear casos de prueba, teniendo en cuenta que podemos definir cuatro niveles de cobertura:
  - Creación de un conjunto de casos de prueba de forma que TODOS LOS ESTADOS sean visitados. El nivel de cobertura es "bajo"
  - Creación de un conjunto de casos de prueba de forma que TODOS LOS EVENTOS sean generados al menos una vez. El nivel de cobertura es "bajo"
  - Creación de un conjunto de casos de prueba de forma que TODOS LOS CAMINOS sean ejecutados al menos una vez. Si el diagrama tiene "bucles", el número de posibles caminos puede ser infinito
  - Creación de un conjunto de casos de prueba de forma que TODAS LAS TRANSICIONES sean ejercitadas al menos una vez. Proporciona un buen nivel de cobertura sin generar un número excesivo de tests.
- Las entradas vendrán dadas por los eventos, y el resultado esperado será el obtenido a través de la secuencia de acciones correspondientes

# Notación gráfica

- El siguiente ejemplo muestra un diagrama de transición de estados de una reserva de billetes de avión

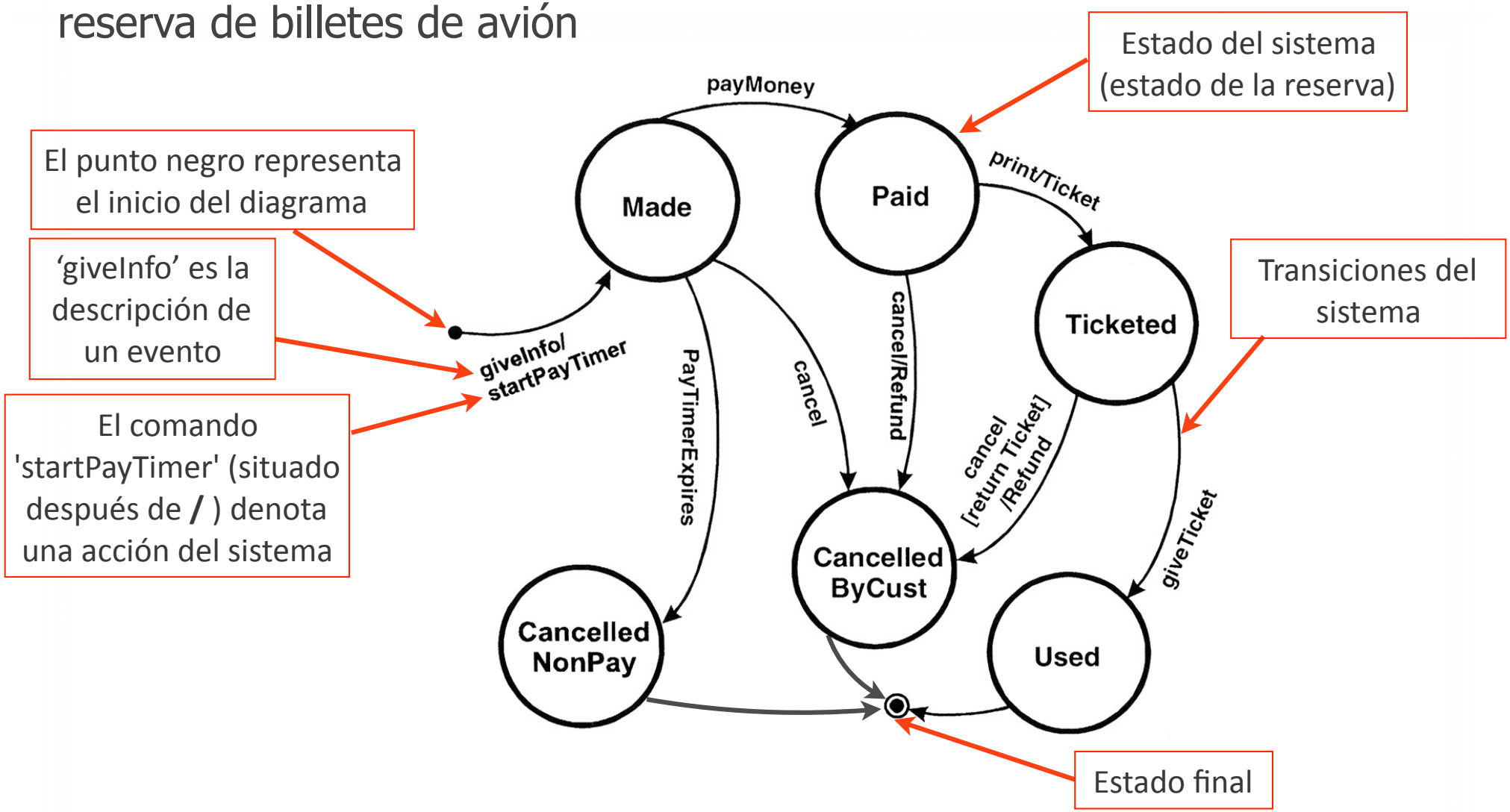
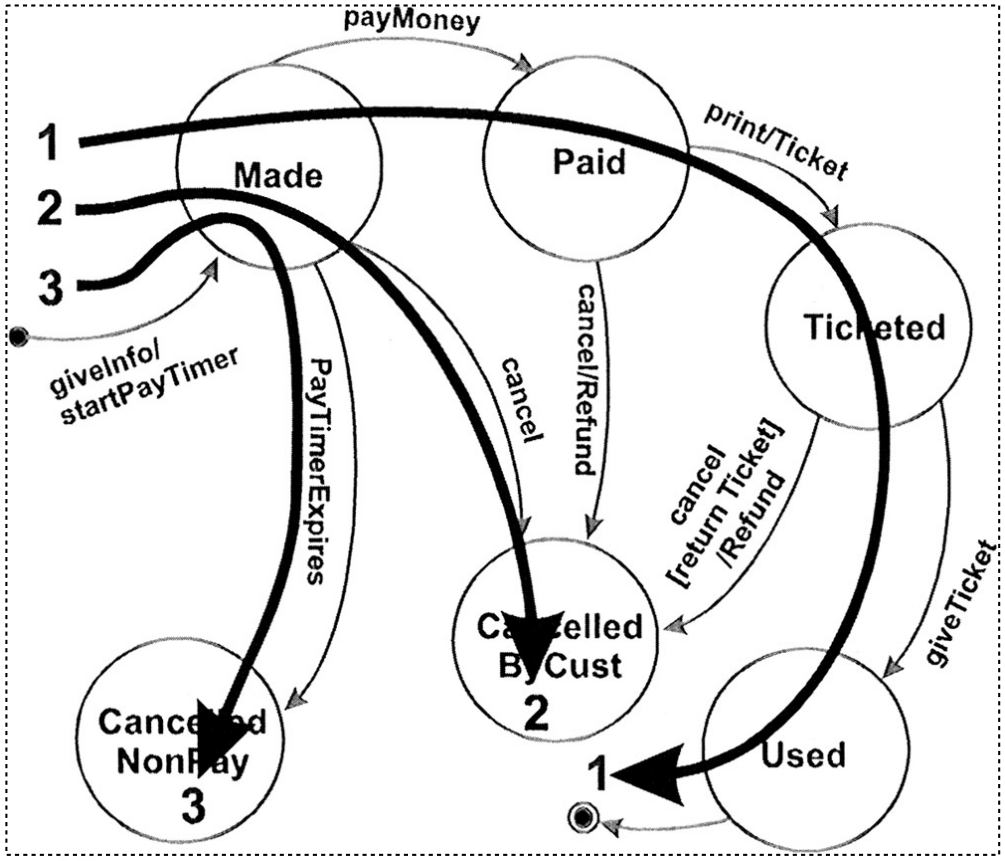
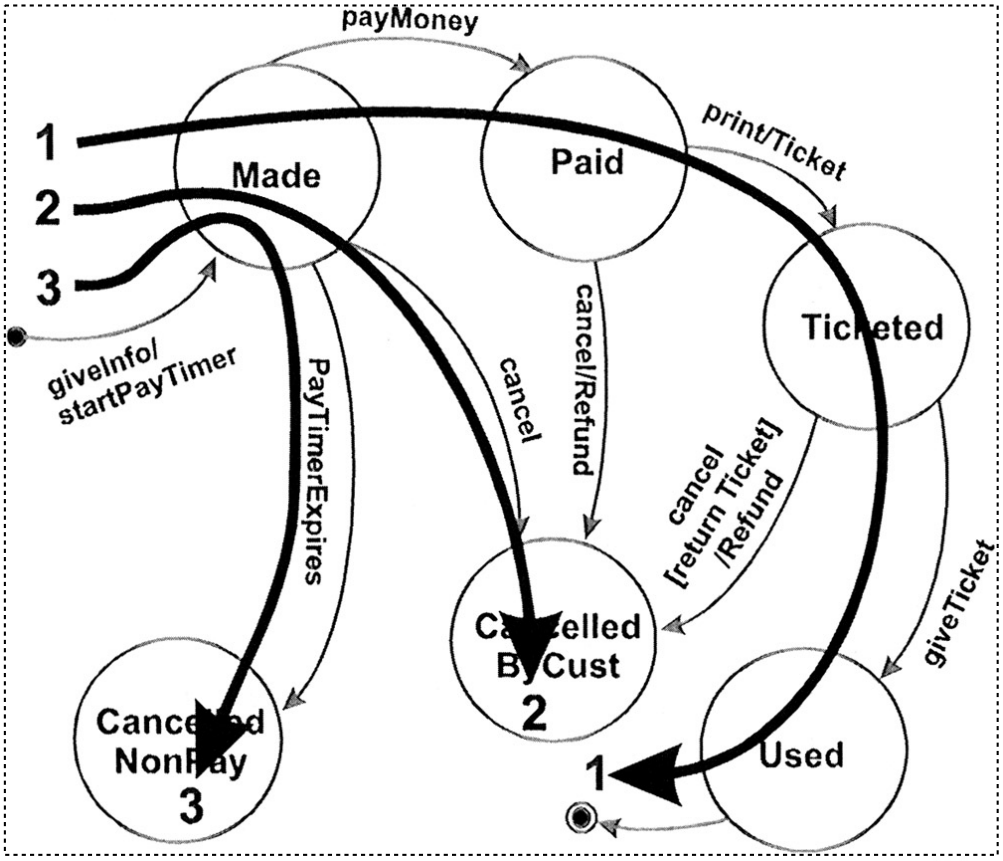



imagen extraída de "A practitioner's guide to software test design". Lee Copeland. Capítulo 7

# Estrategias para generar los tests (I)

- Seleccionamos el conjunto de casos de prueba que recorren TODOS LOS ESTADOS
- Seleccionamos el conjunto de casos de prueba que recorren TODOS LOS EVENTOS




imagenes extraídas de “A practitioner’s guide to software test design”. Lee Copeland. Capítulo 7

# Estrategias para generar los tests (II)

- Conjunto de casos de prueba que recorren TODAS LAS TRANSICIONES
  - Seleccionamos los 5 caminos
  - A continuación tenemos que obtener los casos de prueba de forma que los datos de entrada recorran exactamente todos y cada uno de los caminos

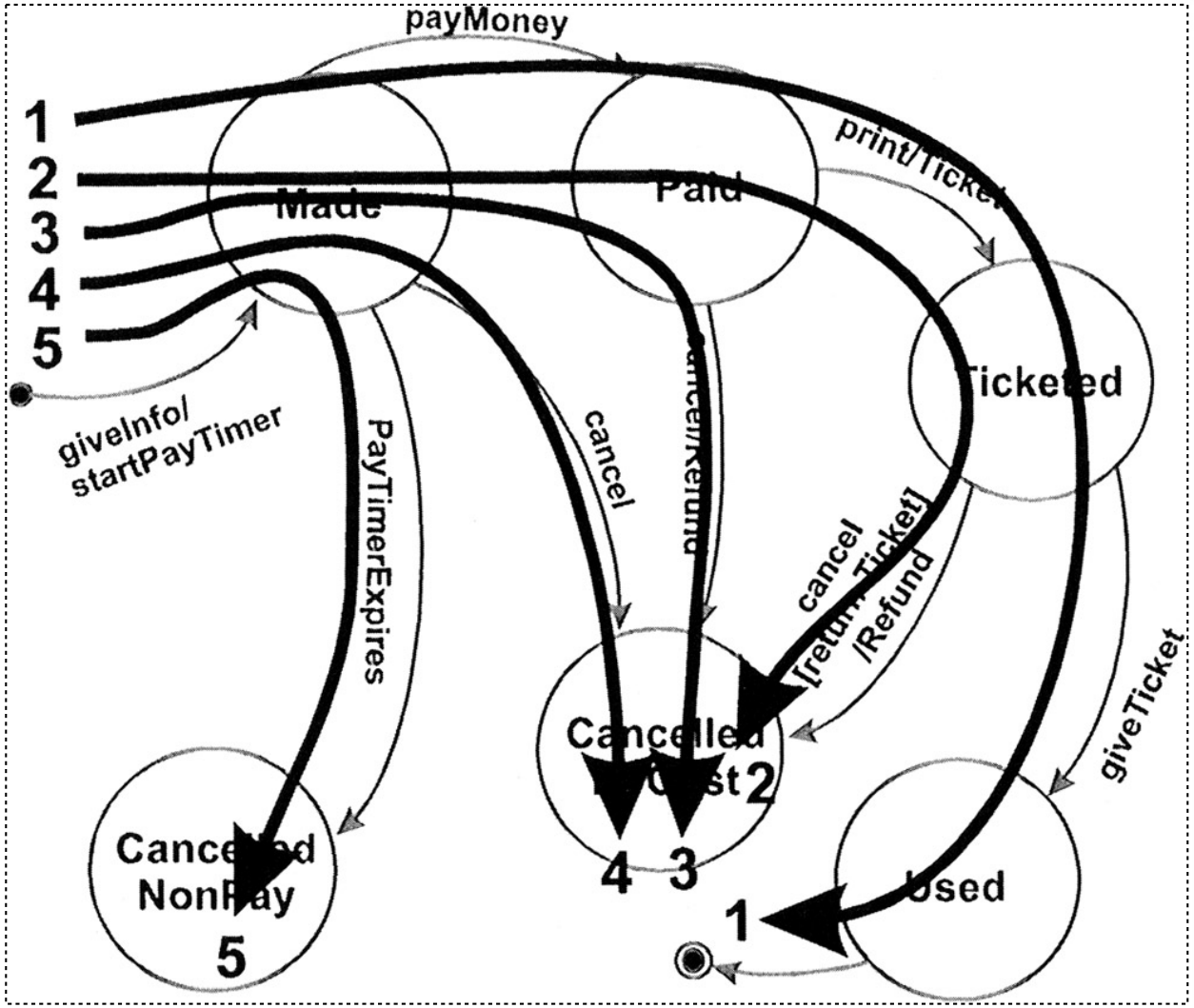



imagen extraída de “A practitioner’s guide to software test design”. Lee Copeland. Capítulo 7


# Ejemplo: sistema de matriculación universidad

- Se trata de probar los procesos de matriculación/anulaciónMatrícula de un alumno en una asignatura. Cuando se crea una asignatura en el sistema, ésta contendrá la siguiente información inicial: identificador de la asignatura, descripción y cuatrimestre. Cada asignatura puede aceptar un número máximo de 10 alumnos matriculados. Podremos matricular a un nuevo alumno (proporcionando su nif) en una asignatura siempre y cuando no se haya superado el número máximo de matrícula. Podremos anular la matrícula de cualquier alumno que haya sido matriculado previamente, proporcionando su nif. Cuando una asignatura ha cubierto el máximo de matriculados, si sigue recibiendo peticiones de matriculación, se creará una una lista de espera, de forma que cualquier alumno que no haya podido ser matriculado, podrá incorporarse a dicha lista a la espera de que haya plazas libres (como consecuencia de la anulación de matrículas).
  - Supondremos que no se va a intentar matricular a un alumno dos veces, ni intentar anular la matrícula de alguien que no está matriculado previamente.
  - Utilizaremos un diagrama de transición de estados para diseñar los casos de prueba

Ver “A practitioner’s guide to software test design”. Lee Copeland. Capítulo 7

# Estados, Eventos, Transiciones

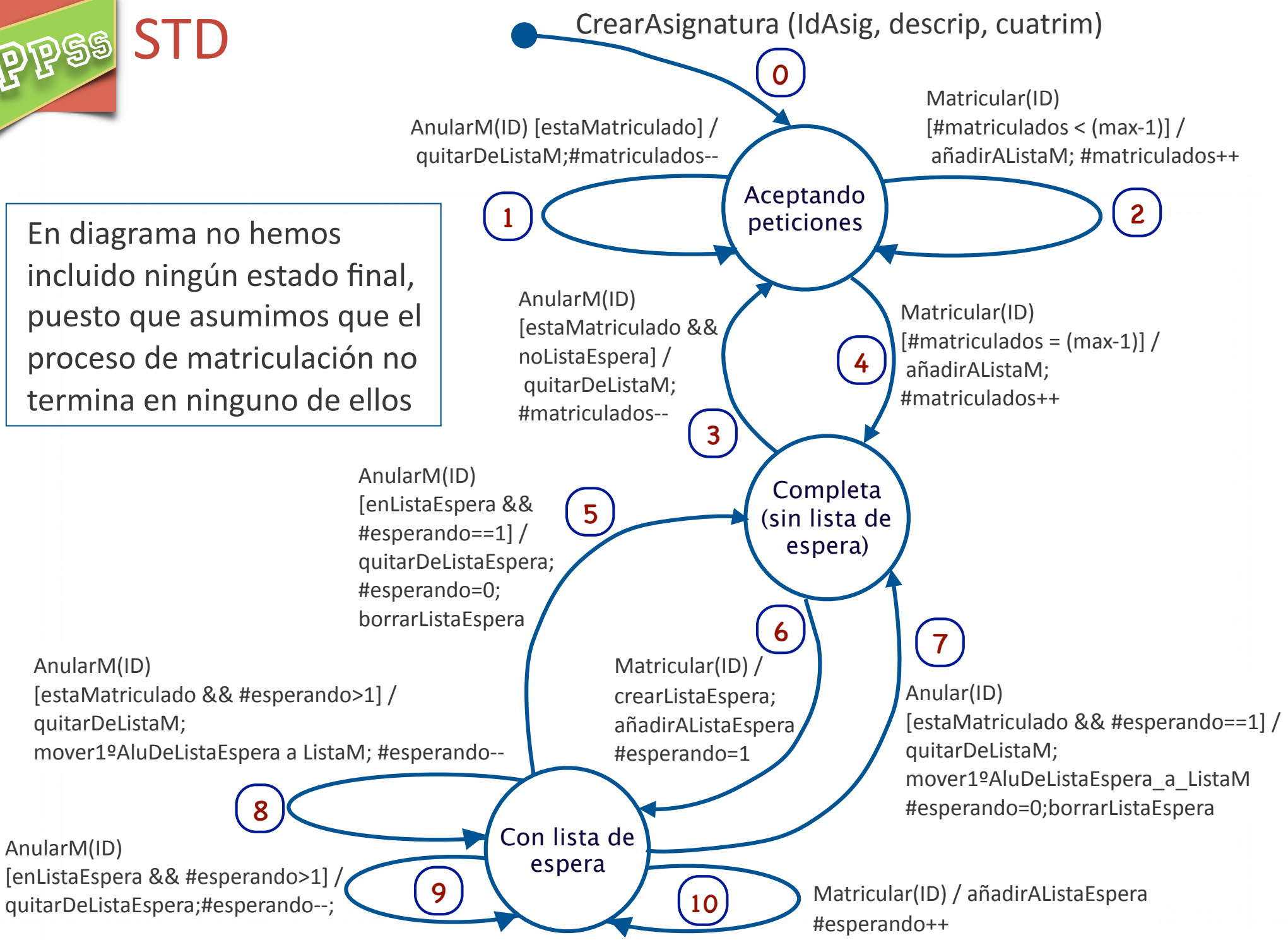
- ¿Qué entidad queremos modelar? Queremos gestionar la matriculación/ anulaciónMatrícula de un alumno en una ASIGNATURA. La asignatura es la entidad que irá cambiando de estado


- ¿Cuáles son los ESTADOS del sistema? La asignatura puede estar básicamente en tres estados:
  - Aceptando peticiones de matriculación
  - Completa (sin lista de espera)
  - Con lista de espera
- EVENTOS del sistema: son entradas que pueden provocar cambios de estado
  - CrearAsignatura(IDasig, descrip, cuatrim), Matricular(ID), AnularM(ID)
- CONDICIONES del sistema (guardas)
  - estaMatriculado, enListaEspera


# Otros elementos del STD

- Acciones:
  - añadirAListaM, quitarDeListaM
  - añadirAListaEspera, quitarDeListaEspera
  - mover1ºAluDeListaEspera_a_listaM
  - borrarListaEspera
- Atributos de los eventos
  - ID: nif alumno
  - max: máximo nº alumnos permitidos
  - #matriculados: nº actual matriculados
  - #esperando: nº actual en espera
- Listas:
  - ListaM: lista de alumnos matric.
  - ListaEspera: lista de alumnos en espera
- Otros símbolos:
  - ++: incrementamos en 1
  - --: decrementamos en 1

# STD




En diagrama no hemos incluido ningún estado final, puesto que asumimos que el proceso de matriculación no termina en ninguno de ellos


# Casos de prueba

- Buscaremos casos de prueba de forma que se cubran todas las transiciones. Un posible conjunto de casos de prueba es el formado por datos de entrada que recorran las siguientes transiciones:
  - 0, 2 (max-1 veces), 4, 3, 1
    - crearAsignatura(34027,2,"pruebas")
    - matricular(00000001A),...matricular(00000009I)
    - matricular(00000010J)
    - anular(00000010J)
    - anular(00000009I)
    - Resultado esperado: listaM(0000001A,...,00000008H), ListaEspera= vacía

(**)

Asignatura: 34027, cuatrimestre: 2, descripción: pruebas
Nif alumnos: 00000001A, 00000002B, ... 00000010J (asumiremos que estos 10 nifs son válidos)

  - 0, 2 (max-1 veces), 4, 6, 5
    - **(**)**
    - matricular(00000011K)
    - anular (00000011K)
    - Resultado esperado: listaM(0000001A,...,00000008H), ListaEspera= vacía
  - 0, 2 (max-1 veces), 4, 6, 10, 9, 7
  - 0, 2 (max-1 veces), 4, 6, 10, 9, 5
  - 0, 2 (max-1 veces), 4, 6, 10, 8

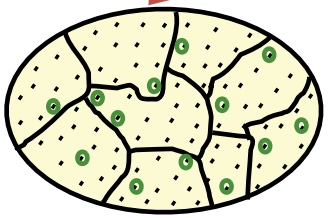
# Y ahora vamos al laboratorio...

## Identificaremos casos de prueba utilizando métodos funcionales

**ESPECIFICACIÓN**

(unidad o sistema)

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



Método de particiones equivalentes

A nivel de unidad

Método de transición de estados

A nivel de sistema

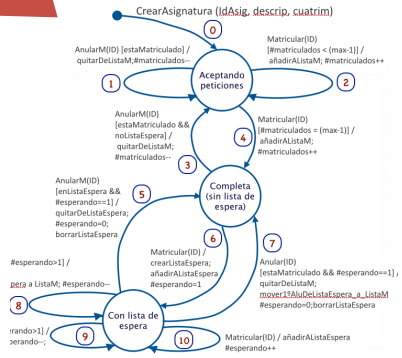



Tabla de casos de prueba

| Identificador CP | Datos Entrada | Resultado Esperado | Resultado Real |
|---|---|---|---|
| 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