PARCIAL 1

# RESUMEN PPSS

Curso 2020-2021

Francisco Javier Pérez Martínez

# Contenido

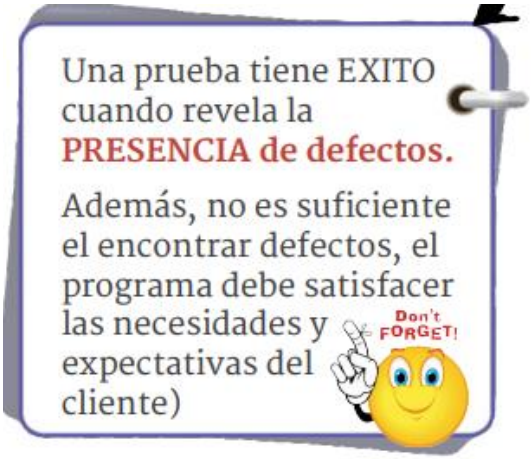
# TEMA 1: Caja Blanca

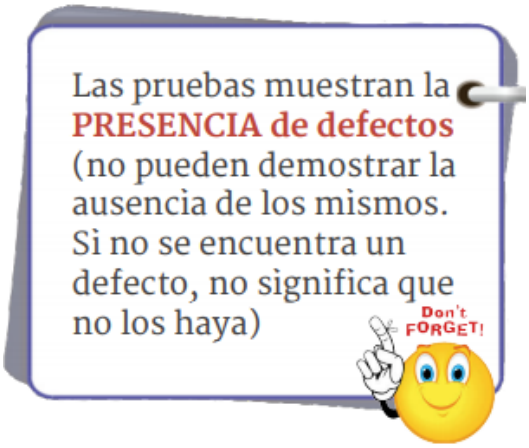
- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de comportamientos programados.

- El conjunto obtenido debe detectar el máximo número posible de defectos en el código, con el mínimo número posible de "filas".

Control flow testing: Método del camino básico

- Paso 1: Análisis de código: Construcción del CFG.
- Paso 2: Selección de caminos: Cálculo de CC y caminos independientes.
- Paso 3. Obtención del conjunto de casos de prueba.



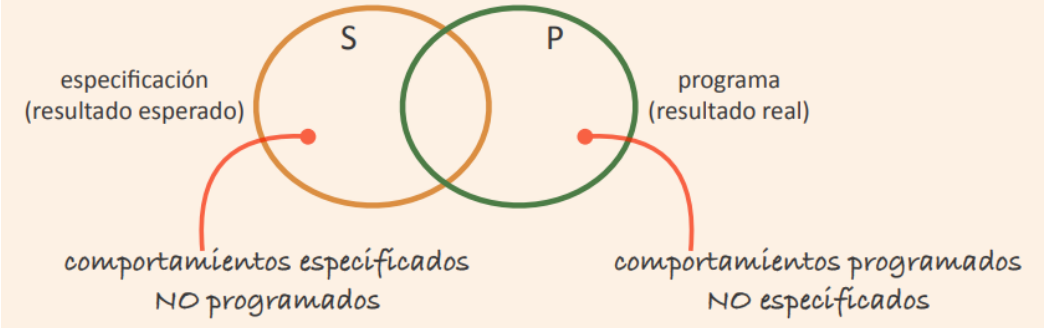




## 1.1 Actividades del proceso de pruebas

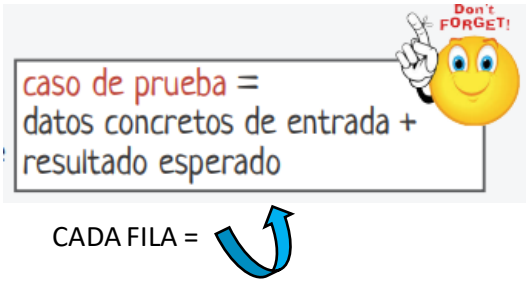
1. Planificación y control de las pruebas
2. Diseño de las pruebas
3. Implementación y ejecución de las pruebas
4. Evaluación del proceso de pruebas y emitir un informe

## 1.2 Pruebas y comportamiento

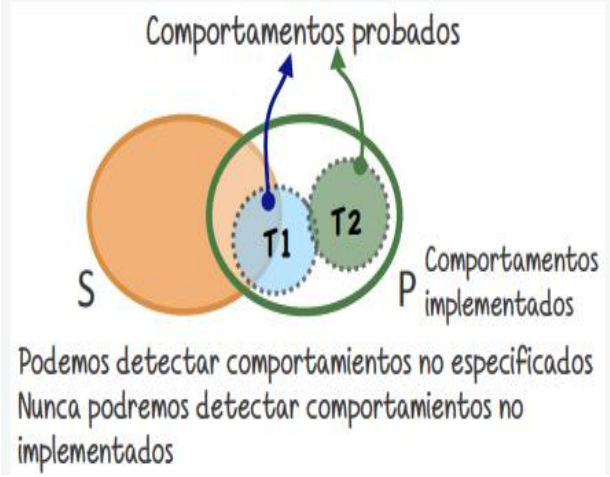
S y P deberían ser idénticos!!!



El objetivo principal al realizar un conjunto de casos de prueba mediante un método de diseño de pruebas es encontrar el máximo número posible de defectos con el mínimo número de casos de prueba.



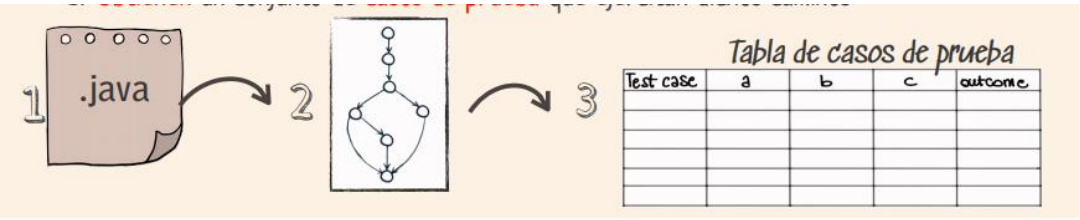

## 1.3 Diseño de casos de prueba (CAJA BLANCA)




- Consiste en determinar los valores de entrada de los casos de prueba a partir de la IMPLEMENTACIÓN.
- El resultado esperado se obtiene SIEMPRE de la especificación.
- Los comportamientos probados podrán estar o no especificados.
- Es esencial conocer conceptos de teoría de grafos para entender bien esta aproximación.

## 1.4 Principios para cualquier método basado en el código

1. Analizan el código y obtienen una representación en forma de GRAFO.
2. Seleccionan un conjunto de caminos en el grafo según algún criterio.
3. Obtienen un conjunto de casos de prueba que ejercitan dichos caminos.



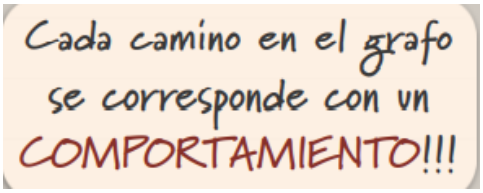

Observaciones:

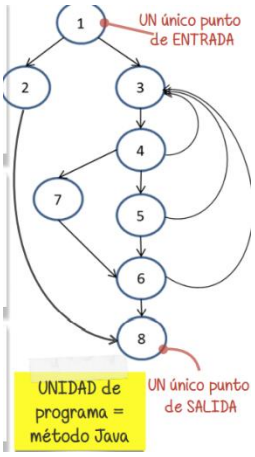
- Dependiendo del método utilizado se obtendrán DIFERENTES conjuntos de casos de prueba. ¡¡¡Pero éste debe ser EFECTIVO y EFICIENTE!!!
- Técnicas o métodos estructurales son costos de aplicar, sólo se usan a nivel de UNIDADES de programa.
- Los métodos estructurales NO pueden DETECTAR TODOS los defectos en el programa.

## 1.5 Grafo de flujo de control (CFG)

Uso de GRAFO DIRIGIDO.

- Cada **nodo** representa una o más sentencias secuenciales y/o una ÚNICA CONDICIÓN. (único punto de entrada y único punto de salida).
  - Cada nodo etiquetado con un entero cuyo valor será único.
  - Anotar a la derecha de un nodo condición.
- Las **aristas** representan el flujo de ejecución entre dos conjuntos de sentencias.
  - Si un nodo contiene una condición etiquetaremos las aristas con T o F.




## 1.6 Método del camino básico

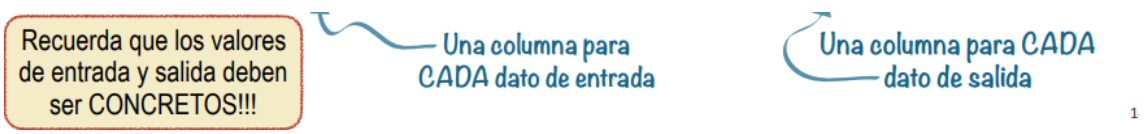
El objetivo de este método es ejecutar TODAS las sentencias del programa, al menos una vez para garantizar así que TODAS las condiciones se ejecutan ya sea V/F.

- El Nº de caminos independientes (CI) determinará el Nº de filas de la tabla.

Pasos:

1. Construir el CFG a partir del código a probar.
2. Calcular la complejidad ciclomática (CC).
3. Obtener los CI del grafo.
4. Determinar los datos de entrada (y salida esperada) de la unidad a probar, ejercitando así todos los CI.

NOTA: el resultado esperado se obtendrá de la ESPECIFICACIÓN de la unidad a probar.




1

### 1.6.1 Formas de calcular CC

- CC = nº de arcos – nº de nodos + 2.
- CC = nº de regiones
- CC = nº de condiciones + 1

¡CUIDADO!: Las dos últimas formas de cálculo son aplicables SOLO si el código es totalmente estructurado (no saltos incondicionales)

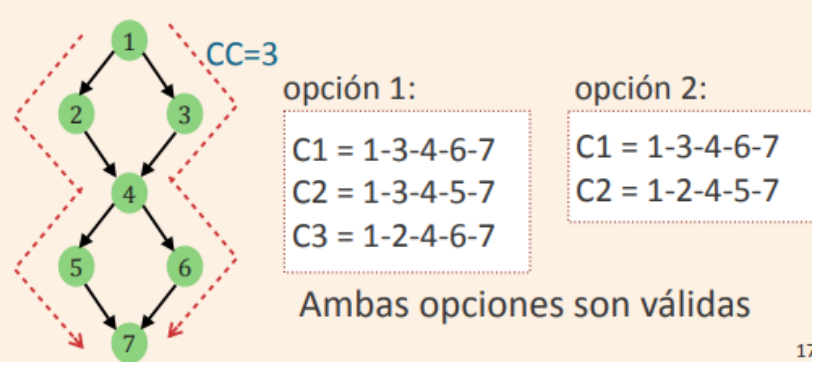
El valor de CC indica el MÁXIMO número de CI en el grafo.

### 1.6.2 Caminos independientes

Buscar como máximo tantos CI como valor obtenido de CC.

- Cada CI contiene un nodo, o bien una arista, que no aparece en los caminos independientes anteriores.
- Con ellos recorremos TODOS los nodos y TODAS las aristas del grafo.

Es posible que con un nº inferior al CC recorramos todos los nodos y aristas.



## P01A

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

**GIT**: Herramienta de gestión de versiones.

**MAVEN**:

- Herramienta automática de construcciones de proyectos java. El "*build script*" se especifica de forma declarativa en el fichero **pom.xml**, en el que encontramos varias partes bien diferenciadas. Los artefactos Maven generados se identifican mediante sus coordenadas.

- Los proyectos Maven requieren una estructura de directorios fija. El código de los tests está físicamente separado del código fuente de la aplicación.

- Maven usa **diferentes ciclos de vida**. Cada ciclo de vida está formado por una **secuencia ordenada de fases**, cada fase puede tener asociadas unas **goals**. El resultado del proceso de construcción Maven puede ser "Build fauilure" o "Build success".

- Cada **fase** puede tener asociadas **cero o más** acciones ejecutables.
- Una **goal** puede asociarse a una fase. Un **plugin** tiene **1 o varias goals**.

**TESTS:**

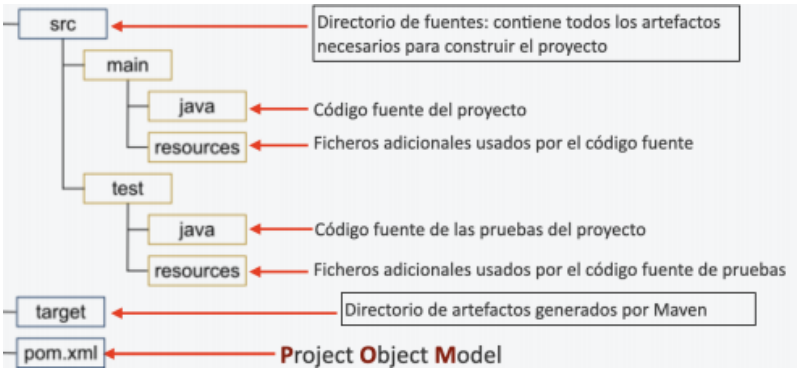
- Un test se basa en un caso de prueba, el cual tiene que ver con el comportamiento especificado del elemento a probar.

- Una vez definido el caso de prueba para un test, éste puede implementarse como un método java anotado con @Test (anotación JUnit). Los tests se compilan y se ejecutan en diferentes momentos durante el proceso de construcción de un proyecto.

- El algoritmo de un test es siempre el mismo. Un test comprueba, dadas unas determinadas entradas sobre el elemento a probar, si su ejecución genera un resultado idéntico al esperado (es decir, se trata de comprobar si el comportamiento especificado en **S coincide con el comportamiento implementado P**).

- Dependiendo de cómo hayamos diseñado los casos de prueba, detectaremos más o menos errores (discrepancias entre el resultado esperado y el resultado real).

**Estructura de directorio de Maven**



## P01B

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

**MÉTODOS DE DISEÑO ESTRUCTURALES**

- Permiten seleccionar un subconjunto de comportamientos a probar a partir del código implementado. Sólo se aplican a nivel de unidad, y son técnicas dinámicas.
- No pueden detectar comportamientos especificados que no han sido implementados.

**MÉTODO DE DISEÑO DEL CAMINO BÁSICO**

- Usa una representación de grafo de flujo de control (CFG).
- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas las líneas del programa, y todas las condiciones en sus vertientes verdadera y falsa, al menos una vez.
- A partir del grafo, el valor de **CC** indica una cota superior del número de casos de prueba a obtener.
- Una vez que obtenemos el conjunto de caminos independientes, hay que proporcionar los casos de prueba que ejerciten dichos caminos (admitiremos que el número de caminos independientes sea ≤ que CC en todos los casos).
- Es posible que haya caminos imposibles o que detectemos comportamientos implementados, pero no especificados.
- Los datos de entrada y los datos de salida esperados siempre deben ser concretos.
- Las entradas de una unidad **no tienen por qué ser** los parámetros de dicha unidad.
- El resultado esperado siempre debemos obtenerlo de la **especificación** de la unidad a probar.

# TEMA 2: Drivers

## 2.1 Automatización de las pruebas

- Se trata de implementar código (**drivers**) para ejecutar los tests de forma automática.
- Implementaremos tantos **drivers** como casos de prueba.
- Cada **driver** invocará a nuestra **SUT** (código a probar) y proporcionará un informe.

## 2.2 Pruebas unitarias

Objetivo: encontrar DEFECTOS en el código de las UNIDADES probadas

La CUESTIÓN fundamental será cómo AISLAR el código de cada unidad a probar

## 2.3 Pruebas de unidad dinámicas: drivers

Ejecutar una unidad de forma AISLADA para poder detectar defectos en el código de dicha unidad.

Utilizando el API JUnit, nos permitirá implementar los drivers y ejecutar los casos de prueba sobre componentes (SUT) de forma automática.

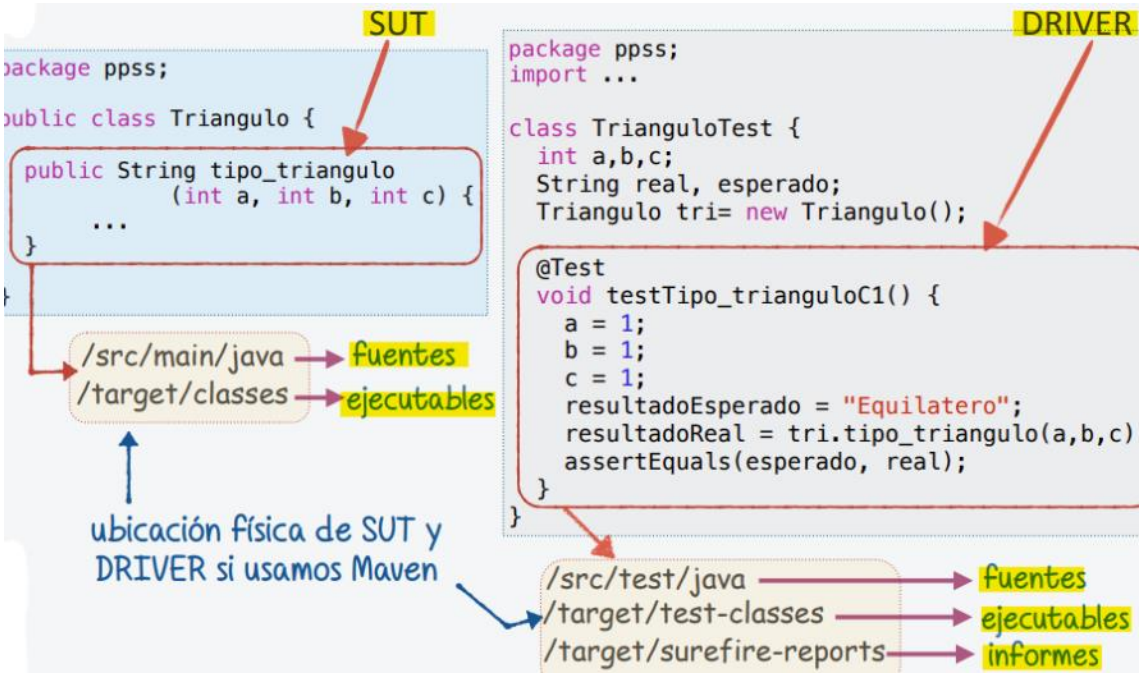
JUnit denomina test (**uno por cada caso de prueba**) a un MÉTODO **sin parámetros**, que devuelve **void**, y está anotado con **@Test.**

Cada método anotado con @Test implementará un driver para un ÚNICO caso de prueba !!!!


No es suficiente con conocer el API JUnit, hay que usarlo correctamente, siguiendo unas normas !!!

## 2.4 Implementación de drivers

El código de pruebas está físicamente separado del código fuente del SUT.




### 2.4.1 Sentencias Assert

- Junit proporciona sentencias (**aserciones**) para determinar el resultado de las pruebas y poder emitir el informe correspondiente.

- Son métodos estáticos, cuyas principales características son:
  - Se utilizan para comparar el resultado esperado con el resultado real.
  - El orden de los parámetros para los métodos assert... es:
    - resultado ESPERADO, resultado REAL [, mensaje opcional]

Todos los métodos "assert" generan una excepción de tipo AssertionFailedError si la aserción no se cumple!!!

Un test puede requerir varias aserciones

### 2.4.2 Agrupación de aserciones

Dado que un test termina en cuanto se lanza la primera excepción (no capturada), en el caso de que nuestro test contenga varias aserciones, usaremos el método **assertAll**, para agruparlas. Este caso, se ejecutan todas, y si alguna falla se lanza la excepción MultipleFailuresError.

```
//Agrupamos las aserciones. SE EJECUTAN TODAS siempre
assertAll("GrupoTestC3",
            ()-> assertArrayEquals(arrayEsperado, coleccion.getColeccion()),
            ()-> assertEquals(numElemEsperado, coleccion.size())
); //Se muestran todos los "fallos" producidos
```

```
---
//No agrupamos las excepciones. Si falla la primera, la segunda NO se ejecuta
assertArrayEquals(arrayEsperado, instance.getColeccion());
assertEquals(numElemEsperado, instance.size());
```

Sólo se ejecuta si assert anterior no

### 2.4.3 Pruebas de excepciones

- Si el resultado esperado es que nuestro SUT lance una excepción, usaremos la aserción **assertThrows().**

```
//Si sut() lanza la excepcion de tipo ExpectedException
//asignamos la excepción a la variable "exception"
ExpectedException exception = assertThrows(ExpectedException.class,
                                    () -> sut(e1,e2));
```

- Si el resultado esperado es que nuestro SUT no lance ninguna excepción, usaremos la aserción **assertDoesNotThrow().**

    - Código más limpio.
    - Evitar test que lanza excepción ya que el resultado será ERROR, en lugar de FAILURE.

### 2.4.5 Anotaciones @BeforeEach, @AfterEach, @BeforeAll, @AfterAll

- **@BeforeEach:** En el caso de que TODOS los **tests** requieran las MISMAS acciones para preparar los datos de entrada

- **@AfterEach**: las acciones comunes a realizar después de la ejecución de CADA **test** (p.e. cerrar sockets)

| ¡¡Estos métodos son **void** SIN parámetros!! |
|---|

- **@BeforeAll** o **@AfterAll**: acciones previas a la ejecución de TODOS los **tests** una ÚNICA VEZ, (o después de ejecutar todos los tests).

| ¡¡Estos métodos estáticos son **void** SIN parámetros!! |
|---|

NO debemos implementar tests cuya ejecución dependa del resultado de ejecutar ningún otro test!!!


### 2.4.6 Etiquetado de los tests: @Tag

Permite etiquetar nuestros tests para FILTRAR su **"descubrimiento"** y **"ejecución".**

- Si anotamos la clase, automáticamente estaremos etiquetando todos sus tests.
- Podemos usar varias "etiquetas" para la clase y/o los tests.
- Nos permiten **DISCRIMINAR** la ejecución de los tests según sus etiquetas.

### 2.4.7 Tests Parametrizados: @ParameterizedTest, @ValueSource

- Si el código de varios tests es idéntico a excepción de los valores concretos del caso de prueba que cada uno implementa, podemos sustituirlos por un test parametrizado anotado con **@ParameterizedTest**

- Si el test parametrizado solamente necesita un parámetro, de tipo primitivo o String, usaremos la anotación **@ValueSource** para indicar los valores para ese parámetro

- Si el método **@ParameterizedTest** requiere más de un parámetro, usaremos **@MethodSource** indicando un nombre de método.

## P02

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

**IMPLEMEMENTACIÓN DE LOS TESTS**

- Es necesario haber diseñado previamente los casos de prueba para poder implementar los drivers.
- El código de los drivers estará en src/test/java, en el mismo paquete que el código a probar. Nuestra SUT será una unidad, por lo tanto, estaremos automatizando pruebas unitarias, usando técnicas dinámicas. Tendremos UN **driver** para CADA **caso de prueba**.
- Debemos cuidar la implementación de los tests, para no introducir errores en los mismos.
- Para compilar los drivers "dependemos" de la libreria JUnit 5, que habrá que incluir en el pom. Antes de compilar el código de los drivers, es imprescindible que se hayan generado con éxito los .class del código a probar (de src/main/java). Es decir, **nunca** haremos pruebas sobre un código que no compile.
- Debes usar correctamente tanto las anotaciones Junit (@) como las sentencias explicadas en clase (Assertions)

**EJECUCIÓN DE LOS TESTS**

- La ejecución de los tests se integra en el proceso de construcción del sistema, a través de MAVEN, (es muy importante tener claro que "acciones" deben llevarse a cabo y en qué orden). La **goal** surefire:test se encargará de invocar a la librería JUnit en la fase "test" para ejecutar los drivers.
- Podemos ser "selectivos" a la hora de ejecutar los drivers, aprovechando la capacidad de Junit5 de etiquetar los tests.
- En cualquier caso, el resultado de la ejecución de los tests siempre será un informe que ponga de manifiesto las discrepancias entre el resultado esperado (especificado), y el real (implementado). Junit nos proporciona un informe con 3 valores posibles para cada test: pass, fault y error.
- Si durante la ejecución de los tests, alguno de ellos falla, el proceso de construcción se DETIENE y termina con un BUILD FAILURE.
- Debes tener claro que comandos Maven debemos usar y qué ficheros genera nuestro proceso de construcción en cada caso.

# TEMA 3: Caja Negra

## 3.1 Diseño de casos de prueba

**Objetivo**: obtener una tabla de casos de prueba a partir del conjunto de comportamientos especificados.

- Podemos detectar comportamientos **especificados**. NUNCA comportamientos **implementados**.
- 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.

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

## 3.2 Método de diseño: Particiones Equivalentes

A partir de la **ESPECIFICACIÓN** identificamos un conjunto de **CLASES** de equivalencia (o partición) para cada una de las entradas y salidas del "elemento" (unidad, componente, sistema) a probar.

- Cada caso de prueba usará un subconjunto de particiones.
- Garantizar que TODAS las particiones de e/s se prueban AL MENOS UNA VEZ.

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.

## 3.3 Identificación de las clases de equivalencia

Antes que nada, hay que recordar que, para conseguir un conjunto de pruebas EFECTIVO y EFICIENTE, tenemos que ser SISTEMÁTICOS a la hora de determinar las particiones de e/s.

Estas particiones se identifican en base a **CONDICIONES** de e/s de la unidad a probar. Las "variables" de e/s no necesariamente se corresponden con "parámetros" de e/s de la unidad. Además, las particiones deben ser **DISJUNTAS** (no comparten elementos).

Paso 1. Identificar las clases de equivalencia para CADA e/s.

- **RANGO de valores** (dentro del rango, fuera de cada uno de los extremos del rango): una clase válida y dos inválidas (#1)
- **NÚMERO N** (entre 1 y N) (ningún valor, + de N) de valores: una clase válida y dos inválidas (#2)
- **CONJUNTO** de valores ($\in$ conjunto, $\notin$ conjunto): una clase válida y una inválida (#3)
- Si **se piensa** que cada uno de los valores de entrada se van a tratar de forma **diferente** por el programa: una clase válida por valor de entrada (#4)
- Situación **DEBE SER**: una clase válida y una inválida (#5)
- Si los elementos de una partición van a ser tratados de forma distinta, **subdividir** la partición en particiones más pequeñas (#6)

Paso 2. Identificar los casos de prueba asignando un **ID ÚNICO** para cada **partición**.


El orden de los pasos importa!!

- Hasta que todas las clases válidas no estén probadas, escribir un nuevo caso de prueba por cada clase válida.
- Hasta que todas las clases inválidas no estén probadas, escribir un nuevo caso de prueba por cada clase inválida.
- Elegir un valor concreto para cada partición.

El resultado de este proceso será una TABLA con tantas FILAS como CASOS DE PRUEBA.

## P03

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

### MÉTODOS DE DISEÑO DE CAJA NEGRA

- Permiten seleccionar de forma SISTEMÁTICA un subconjunto de comportamientos a probar a partir de la especificación. Se aplican en cualquier nivel de pruebas (unitarias, integración, sistema, aceptación), y son técnicas dinámicas.
- El conjunto de casos de prueba obtenidos será **eficiente** y **efectivo** (exactamente igual que si aplicamos métodos de diseño de caja blanca, de hecho, la estructura de la tabla obtenida será idéntica).
- Pueden aplicarse sin necesidad de implementar el código (podemos obtener la tabla de casos de prueba mucho antes de implementar, aunque no podremos detectar defectos sin ejecutar el código de la unidad a probar).
- No pueden detectar comportamientos implementados, pero no especificados.
- A diferencia de los métodos de caja blanca, pueden aplicarse no solamente a unidades sino a "elementos" "más grandes" (con más líneas de código): pruebas de integración, pruebas del sistema y pruebas de aceptación

### MÉTODO DE PARTICIONES EQUIVALENTES

- Es imprescindible obtener las entradas y salidas de la unidad a probar para poder aplicar correctamente el método. Este paso es igual de imprescindible si aplicamos un método de diseño de pruebas de caja blanca., ya que de ello dependerá la estructura de la tabla de casos de prueba obtenida.
- Cada entrada y salida de la unidad a probar se particiona en clases de equivalencia (todos los valores de una partición de entrada tendrán su imagen en la misma partición de salida). Las **particiones** pueden realizarse sobre cada entrada por separado, o sobre agrupaciones de las entradas, dependiendo de si el carácter valido/inválido de una partición de una entrada, depende de otra de las entradas. Cada partición (tanto de entrada como de salida, agrupadas o no) se etiqueta como válida o inválida. Todas las particiones deben ser disjuntas.

- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas y cada una de las particiones, y que todas las particiones inválidas se prueban de una en una **(sólo puede haber una partición inválida de entrada en cada caso de prueba)**. Para ello es importante seguir un orden a la hora de combinar las particiones: primero las válidas, y después las inválidas, de una en una).
- Cada caso de prueba será una selección de un comportamiento especificado. Puede ocurrir que la especificación sea incompleta, de forma que, al hacer las particiones, no podamos determinar el resultado esperado. En ese caso debemos poner un "**interrogante**" como resultado esperado. El tester NO debe completar/cambiar la especificación.