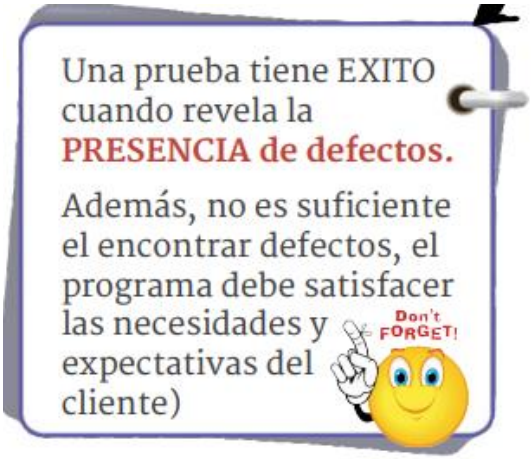
TEMA 1: Caja Blanca

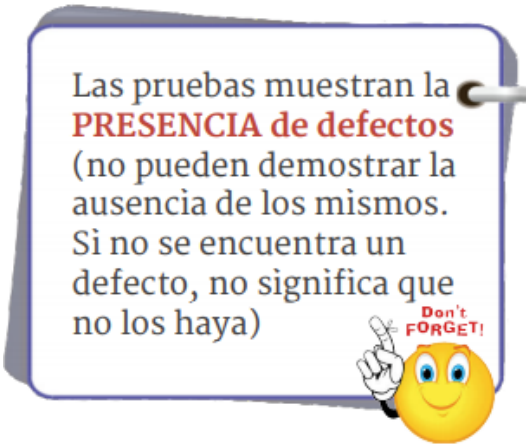
- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de comportamientos programados.

- 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.



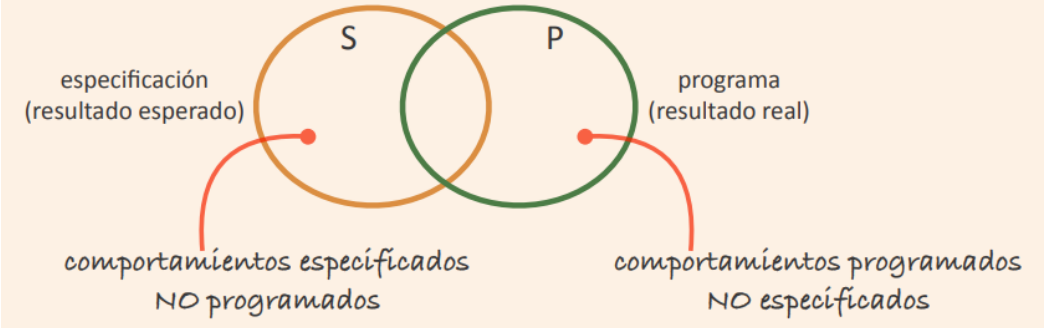


1.1 Actividades del proceso de pruebas

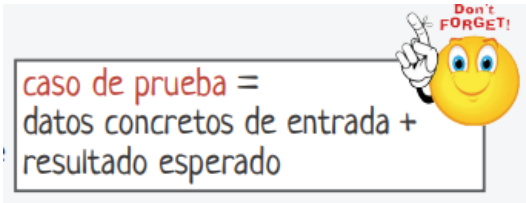
1. Planificación y control de las pruebas
2. Diseño de las pruebas
3. Implementación y ejecución de las pruebas
4. Evaluación del proceso de pruebas y emitir un informe

1.2 Pruebas y comportamiento

S y P deberían ser idénticos!!!

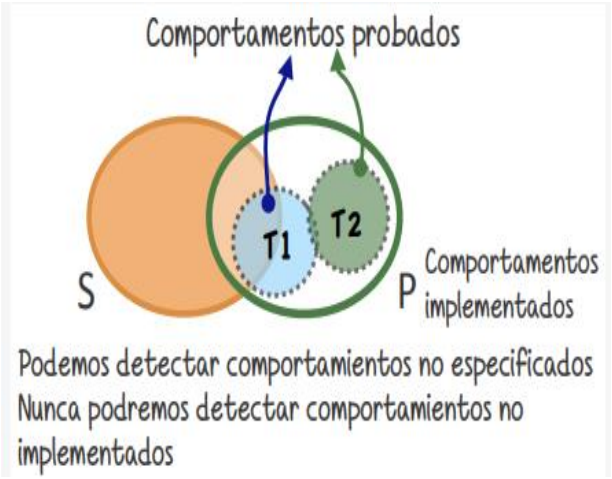


El objetivo principal al realizar un conjunto de casos de prueba mediante un método de diseño de pruebas es encontrar el máximo número posible de defectos con el mínimo número de casos de prueba.



CADA FILA =

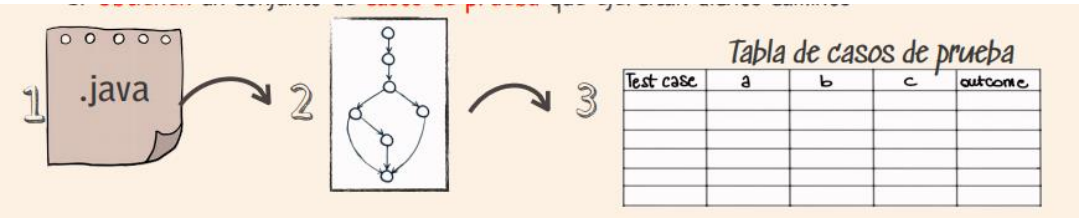
1.3 Diseño de casos de prueba (CAJA BLANCA)



- 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.



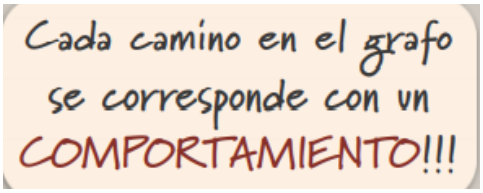
Observaciones:

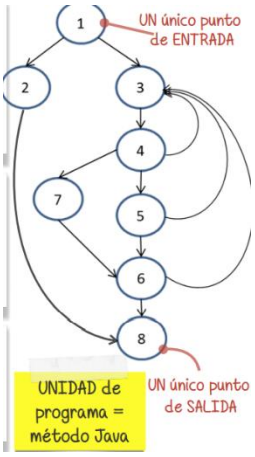
- Dependiendo del método utilizado se obtendrán DIFERENTES conjuntos de casos de prueba. ¡¡¡Pero éste debe ser EFECTIVO y EFICIENTE!!!
- Técnicas o métodos estructurales son costos de aplicar, sólo se usan a nivel de UNIDADES de programa.
- Los métodos estructurales NO pueden DETECTAR TODOS los defectos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

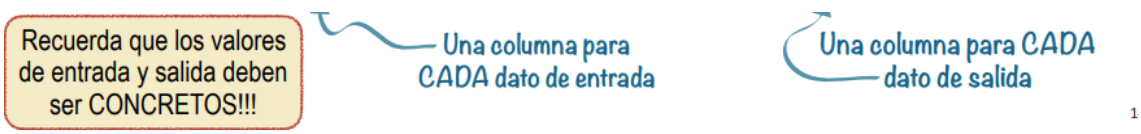
El objetivo de este método es ejecutar TODAS las sentencias del programa, al menos una vez para garantizar así que TODAS las condiciones se ejecutan ya sea V/F.

- 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)

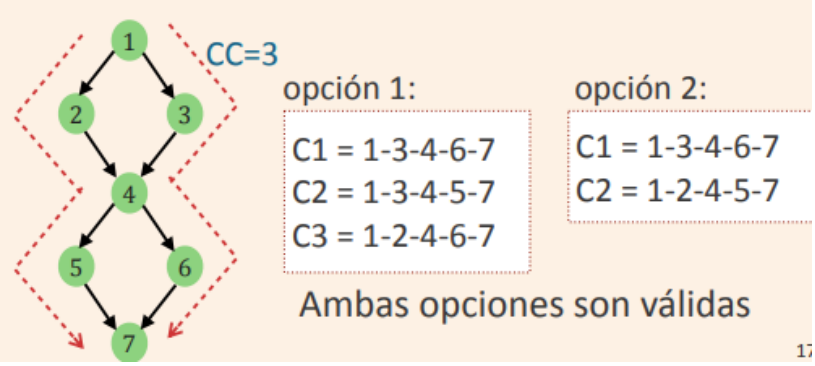
El valor de CC indica el MÁXIMO número de CI en el grafo.

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:

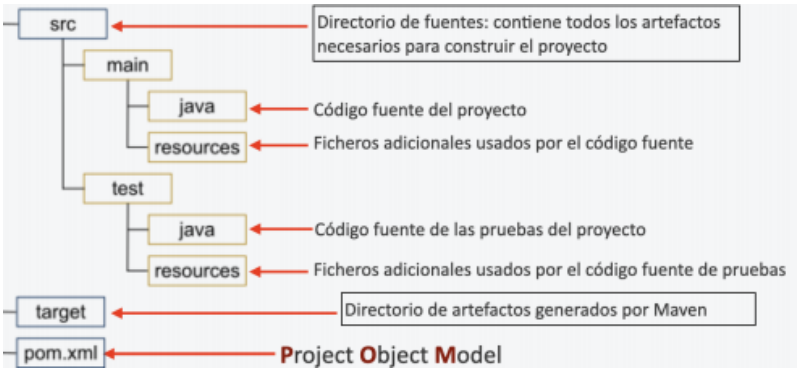
- Un test se basa en un caso de prueba, el cual tiene que ver con el comportamiento especificado del elemento a probar.

- 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.

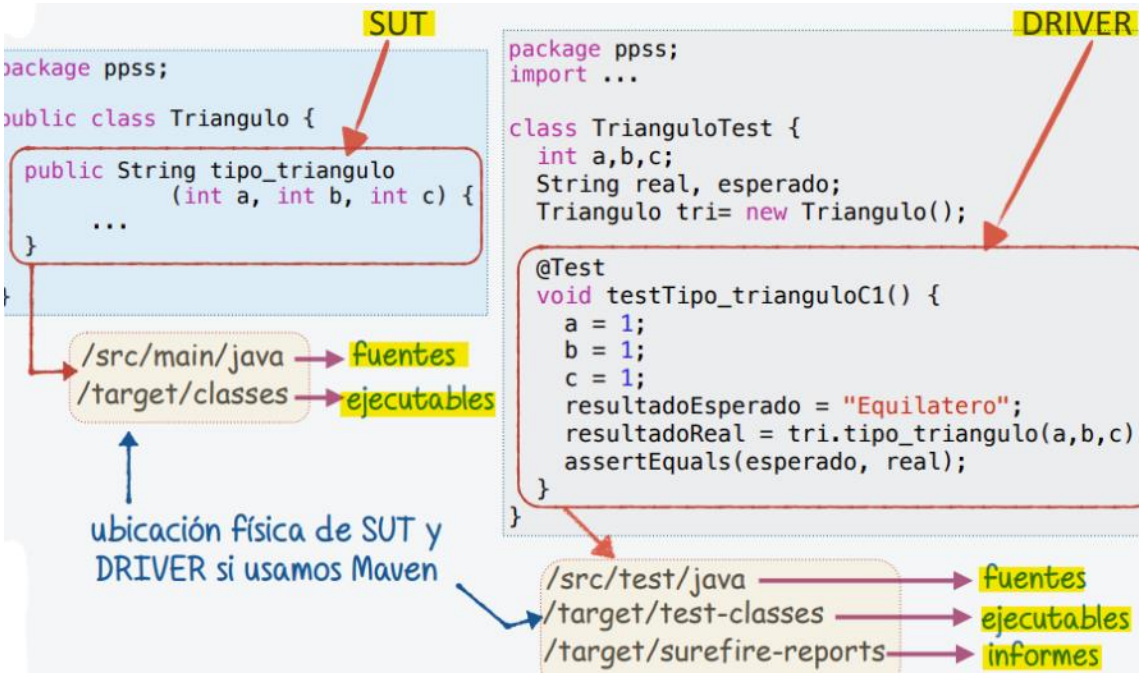
JUnit denomina test (**uno por cada caso de prueba**) a un MÉTODO **sin parámetros**, que devuelve **void**, y está anotado con **@Test.**

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

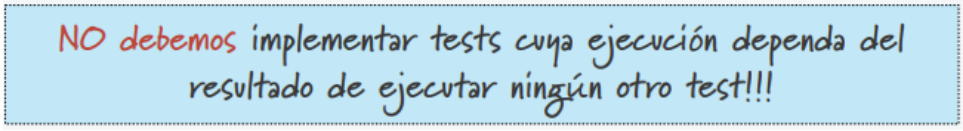
- **@BeforeEach:** En el caso de que TODOS los **tests** requieran las MISMAS acciones para preparar los datos de entrada

- **@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!!



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.

TEMA 4: Dependencias Externas (1)

Pruebas unitarias: implementación de drivers utilizando verificación basada en el estado

Conceptos de código testable y "seam"

Proceso para aislar la unidad de sus dependencias externas (control de entradas indirectas):

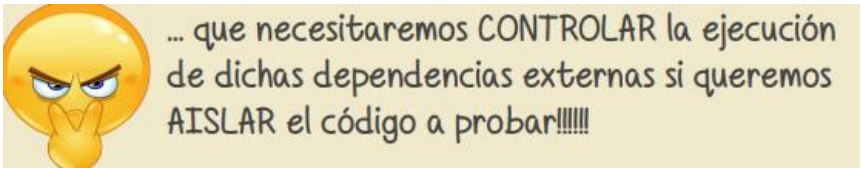
- Paso 1: Identificación de dependencias externas
- Paso 2: Refactorización de la unidad (sólo si es necesario) para conseguir inyectar los dobles de las dependencias externas
- Paso 3: Control de las dependencias externas: implementamos un doble (**stub**) para controlar las entradas indirectas al SUT
- Paso 4: Implementación del driver utilizando verificación basada en el estado

4.1 Pruebas unitarias y dependencias externas

Las pruebas de unidad dinámicas requieren ejecutar cada unidad (SUT: System Under Test) de forma AISLADA para poder detectar defectos (**bugs**) en dicha unidad.

SUT: Código que queremos probar (método Java).

¿Qué ocurre si desde SUT se invoca a otras unidades?



4.2 La Regla "DE ORO" para realizar las pruebas

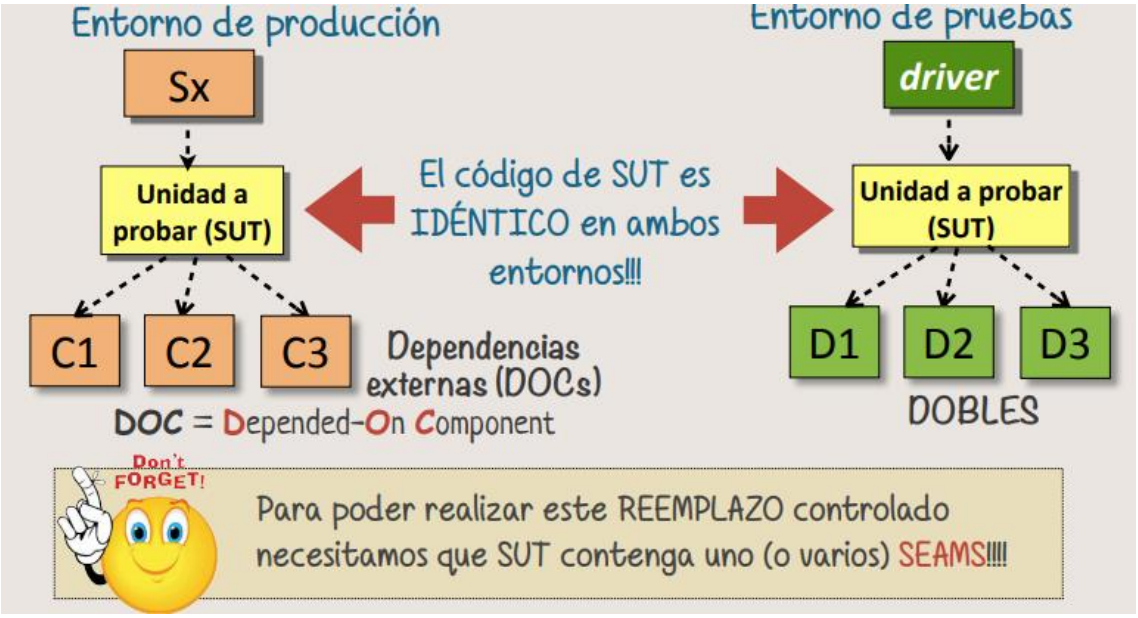
SUT exactamente el mismo código que se utilizará en producción.

No alterar circunstancialmente/temporalmente el código de SUT de ninguna forma a la hora de realizar las pruebas.

4.3 Código testable y control de dependencias

Código testable: permite que un componente sea fácilmente probado de forma AISLADA.

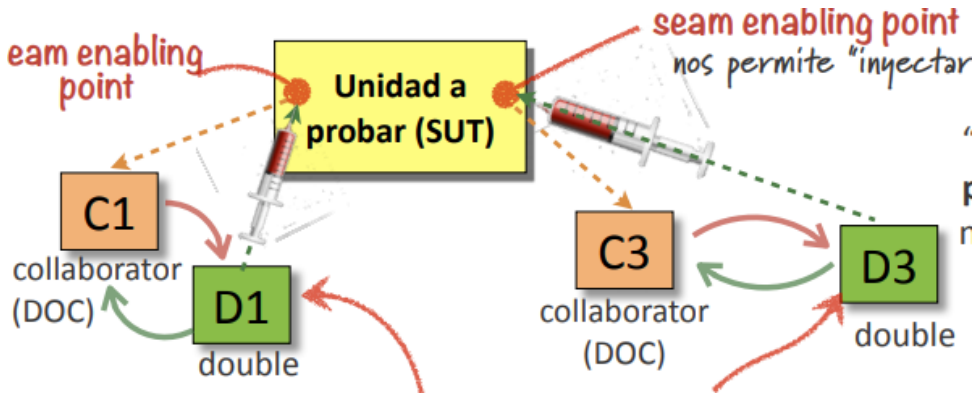
- Para ello, debemos CONTROLAR sus **DEPENDENCIAS externas**. (COLABORADORES o DOCs)
- Dependencia externa: objeto con quién interactúa nuestro SUT y sobre el que no tenemos ningún control.



4.4 SEAM

Para poder conseguir un sean en nuestro código PUEDE que necesitemos **REFACTORIZAR** nuestro SUT.

DOBLES: reemplazos controlados de los colaborados del sistema utilizados durante las pruebas para aislar el código de SUT.



4.4.1 Cómo identificar un SEAM

Imaginemos que tenemos la línea -> myCell.recalculate();

Si no conocemos el tipo de objeto "myCell", **no podemos saber** a que método se invocará.

Si **podemos cambiar el método** que se invocará desde esta línea **SIN** alterar el código que la unidad que la contiene, entonces esta línea de código es un **SEAM.**

Cada SEAM debe tener un "punto de inyección", que, durante las pruebas, nos permite reemplazar cada dependencia externa por su doble (SIN alterar el código de SUT).

4.5 Pasos a seguir para automatizar las pruebas

1. Identificar las dependencias externas de nuestro SUT.
2. Asegurarnos que nuestro **SUT** es **TESTABLE** (probado de forma aislada, debe contener un SEAM). Se debe poder realizar un reemplazo controlado de cada **DOC** por su **DOBLE** sin modificar su código. (REFACTORIZAR)
3. Implementación **DOBLE** que reemplazará el código real de cada DOC.
4. Implementar los DRIVERS correspondientes.
   Verificación basada en el ESTADO:
   a. Sólo estamos interesados en comprobar el estado resultante de la invocación de nuestro SUT.
   Verificación basada en el COMPORTAMIENTO:
   b. Nos interesa, además, verificar que las interacciones entre nuestro SUT y las dependencias externas se realizar correctamente.

NO queremos ejecutar los tests sobre la implementación real de un método, solamente nos interesa controlar **el valor que devuelve** este método.

4.6 EL SUT DEBE SER TESTABLE

Para que sea testable, debe contener un SEAM. Y sino lo es, lo REFACTORIZAMOS.

En Java:

- El DOBLE debe **IMPLEMENTAR** la misma **INTERFAZ** que el DOC o debe **EXTENDER** la misma **CLASE** que éste.
- SUT testable si podemos "inyectar" el doble en nuestra SUT durante las pruebas:
  - Como un **parámetro** de nuestra SUT.
  - A través del **constructor** de la clase que contiene nuestra SUT.
  - A través de un **método setter** de la clase que contiene nuestra SUT.
  - A través de un método de **factoría local**, o **una clase factoria.**

- SUT No testable, entonces REFACTORIZAR para poder inyectar el doble teniendo en cuente que:

    - Si añadimos **un parámetro**, estamos OBLIGANDO a que cualquier código cliente de nuestra SUT tenga que CONOCER dicha dependencia ANTES de invocar a nuestra SUT.

    - Si añadimos un parámetro al **constructor** de nuestra SUT, (**o un setter**) estamos OBLIGANDOS a declarar la dependencia como un atributo de la clase que contiene nuestro SUT.

    - No podremos añadir un **método setter** si el constructor realizase alguna acción significativa sobre nuestra dependencia asumiendo que no se ejecutarán acciones intermedias entre la invocación al constructor y al setter.

    - Si usamos un método de **factoría local**, no se ven afectados, ni los clientes de nuestro SUT, ni la estructura de la clase, aunque alteramos el comportamiento de la clase que contiene nuestro SUT al añadir un nuevo método. Añadir una **clase factoría** puede ser innecesario en producción.

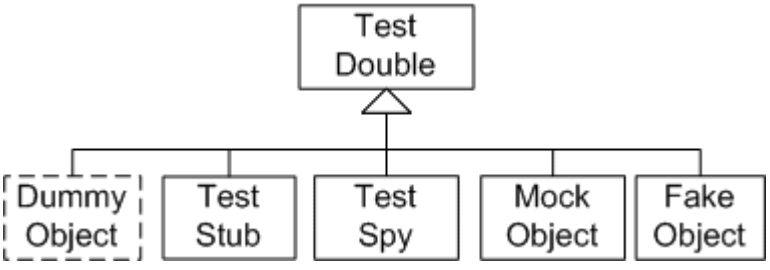
4.7 Implementar el DOBLE

Test Double: objeto o componente utilizado para sustituir al objeto o componente real durante las pruebas.

Test Stub: objeto que actúa como un punto de **CONTROL** para entregar ENTRADAS INDIRECTAS al SUT, cuando se invoca a alguno de los métodos de dicho stub (Utiliza verif. basada en ESTADO).

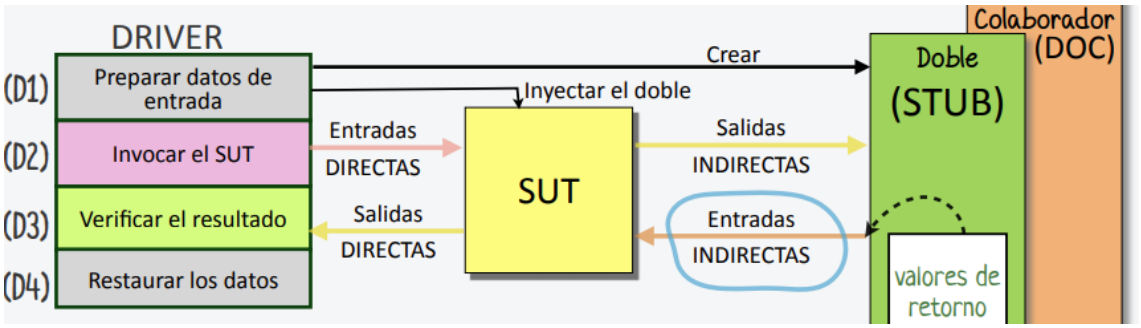
Test Spy: acceder a información oculta, comprobar si se ha invocado un determinado método dentro del SUT.

Mock Objet: objeto que actúa como punto de observación para las SALIDAS INDIRECTAS del SUT. (devuelve info o nada). Registra las llamadas recibidas del SUT, comparando las reales con las expectativas, sino coinciden falla el test (Utiliza verif. basada en el comportamiento).

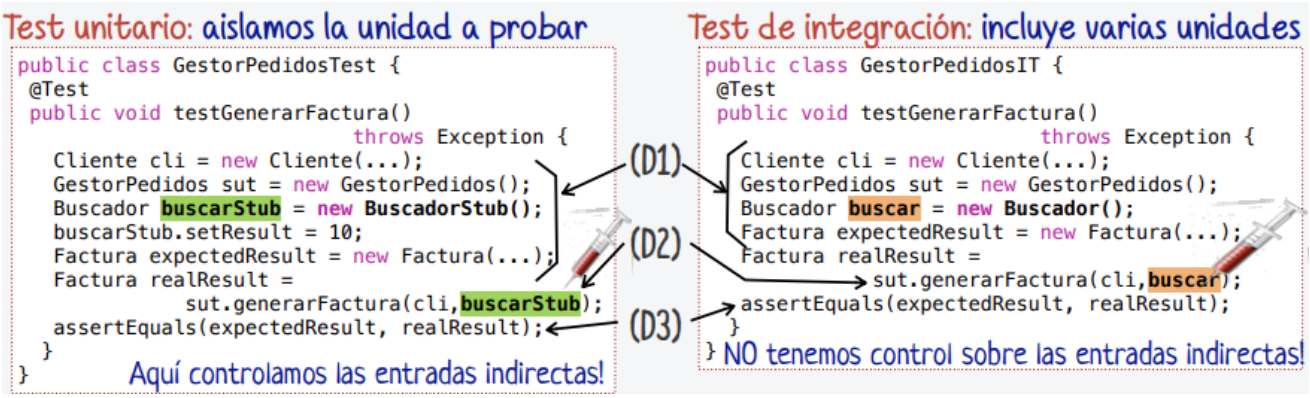


4.8 Implementación del DRIVER

Un **STUB** es un objeto que reemplaza el componente real (**DOC**) del cual depende el código del SUT, para que éste pueda controlar las **entradas indirectas** provenientes de dicha dependencia (valores de retorno, actualización de parámetros o excepciones lanzadas).



El **DRIVER** crea el doble(stub) y lo **inyecta** para que sea invocado por nuestro SUT. Tantos stubs como DOCs necesitemos controlar.




```
public Factura generarFactura(Cliente cli, Buscador buscar)
                                throws FacturaException {
    Factura factura;
    int numElems = buscar.elemPendientes(cli);
    if (numElems>0) {
       //código para generar la factura
      factura = ...;
    } else {
       throw new FacturaException("No hay ...");
    }
    return factura;
}
```

Los mismos tests ejecutan el MISMO CÓDIGO!!! SUT

Los **dobles** y los **drivers** siempre se implementarán en **/src/test/java**

4.9 Test unitario y Test de integración

En un test unitario ejecutamos nuestro SUT **controlando** su dependencia externa. En cambio, en un test de integración ejecutamos nuestro SUT **sin ningún control** de su dependencia externa.

- Test unitario -> *¡¡Ejecutamos el código de nuestro **DOBLE** durante las pruebas!!*
- Test de integración -> *¡¡Ejecutamos el código **REAL** de nuestro DOC durante las pruebas!!*

P04

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

DEPENDENCIAS EXTERNAS

- Cuando realizamos pruebas unitarias la cuestión fundamental es AISLAR la unidad a probar para garantizar que, si encontramos evidencias de DEFECTOS, éstos se van a encontrar "dentro" de dicha unidad...
- Para aislar la unidad a probar, tenemos que controlar sus **entradas indirectas**, proporcionadas por sus colaboradores o dependencias externas. Dicho control se realiza a través de los dobles de dichos colaboradores
- Durante las pruebas realizaremos un reemplazo controlado de las dependencias externas por sus dobles de forma que el código a probar (SUT) será **IDÉNTICO** al código de producción.
- Un **DOBLE** siempre heredará de la clase que contiene nuestro SUT, o implementará su misma interfaz. y será inyectado en la unidad a probar durante las pruebas. Hay varios tipos de dobles, concretamente usaremos **STUBS**.
- Para poder inyectar el doble durante las pruebas, es posible que tengamos que REFACTORIZAR nuestro SUT, para proporcionar un "seam enabling point".

IMPLEMEMENTACIÓN DE LOS TESTS

- El **driver** debe, durante la preparación de los datos, crear los dobles (uno por cada dependencia externa), y debe inyectar dichos dobles en la unidad a probar a través de uno de los "enabling seam points" de nuestro SUT.

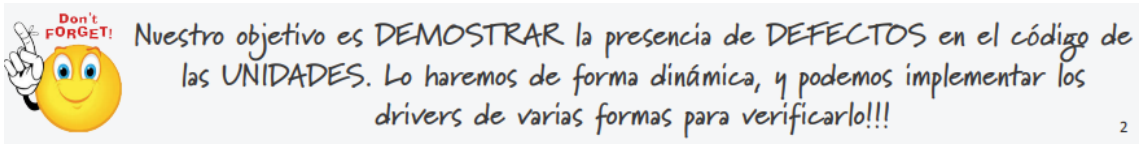
- A continuación, el driver ejecutará nuestro SUT (pasándole **las entradas DIRECTAS** del SUT, mientras que **las entradas INDIRECTAS** las obtendrá de los **STUBS**). El driver comparará el **resultado real obtenido** y finalmente generará un **informe**.

- Dado que el **informe de pruebas** dependerá exclusivamente del **ESTADO** resultante de la ejecución de nuestro SUT, nuestro driver estará realizando una **VERIFICACIÓN BASADA EN EL ESTADO**...

TEMA 5: Dependencias externas (2)

Verificación basada en el ESTADO → implementamos el driver considerando nuestro SUT como una caja negra. (**stub**)

Verificación basada en el COMPORTAMIENTO → implementamos el driver teniendo en cuenta la interacción de nuestro SUT con sus dependencias externas. (**mock**)



Frameworks para implementar dobles

- Podemos utilizar **frameworks** para automatizar los dobles de nuestras pruebas, pero esto conlleva escribir código adicional que a su vez puede contener errores.
    - Genera una implementación configurable de las dobles "on the fly", generando el bytecode necesario.
    - Configuramos los dobles para controlar las entradas indirectas a nuestro SUT (stubs) o para registrar o verificar las salidas indirectas de nuestro SUT (spies y mocks).
    - **Inyectaremos** los dobles en la unidad a probar antes de invocarla.
- Tienen defensores y detractores.
    - Suelen tergiversar la terminología que hemos visto sobre dobles.
    - La verif. Basada en el **comportamiento** (verf. Salidas indirectas) genera un riesgo de que los tests tengan un alto nivel de acoplamiento con los detalles de implementación (**tests frágiles y difíciles de mantener**).
- Frameworks: **EasyMock**, Mockito, JMockit, PowerMock.

EasyMock y Tipos de Dobles

Para crear STUBS: usar el método estático **EasyMock.niceMock(Clase.class)** → devuelve un doble para la clase o interfaz "Clase.class"

No verificamos el orden de las invocaciones de nuestro SUT al doble ya que realizamos una verificación basada en el estado. Sólo **controlamos las entradas indirectas** a nuestro SUT.
EasyMock implementa un **doble** para cada método de la clase. Si no se programan las expectativas, por defecto devuelve 0, null o false. (permite invocaciones no programadas).
El objeto **niceMock** verifica que el SUT invoca al doble usando los parámetros especificados. Métodos anyObject(), anyInt(), etc para no tener encuenta lo anterior.
Para programar las expectativas usaremos el método estático **EasyMock.expect().** Después de programar las expectativas ACTIVAR siempre el stub usando el método **replay().**
Podemos hacer que un stub devuelva un determinado resultado independientemente de los valores de entrada.
Después de invocar a nuestro SUT, SIEMPRE debemos verificar que nuestra SUT ha invocado a los mocks usando **EasyMock.verify()**

```
public class GestorPedidos {
  public Factura generarFactura(Cliente cli, Buscador bus) throws FacturaException {
    Factura factura = new Factura();
    int numElems = bus.elemPendientes(cli);
    if (numElems>0) {
       //código para generar la factura
       factura = ...;
    } else {
        throw new FacturaException("No hay nada pendiente de facturar");
    }
    return factura;
  }
}
```

dependencia externa

SUT TESTABLE!!!

DRIVER

```
public class GestorPedidosTest {
 @Test
 public void testGenerarFactura() throws Exception {
   Cliente cli = new Cliente(...);
   GestorPedidos sut = new GestorPedidos();

   Buscador stub = EasyMock.niceMock(Buscador.class);
   EasyMock.expect(stub.elemPendientes(anyObject())
                       .andStubReturn(10);
   EasyMock.replay(stub);

   Factura expResult = new Factura(...);
   Factura result = sut.generarFactura(cli,stub);
   assertEquals(expResult, result);
  }
}
```

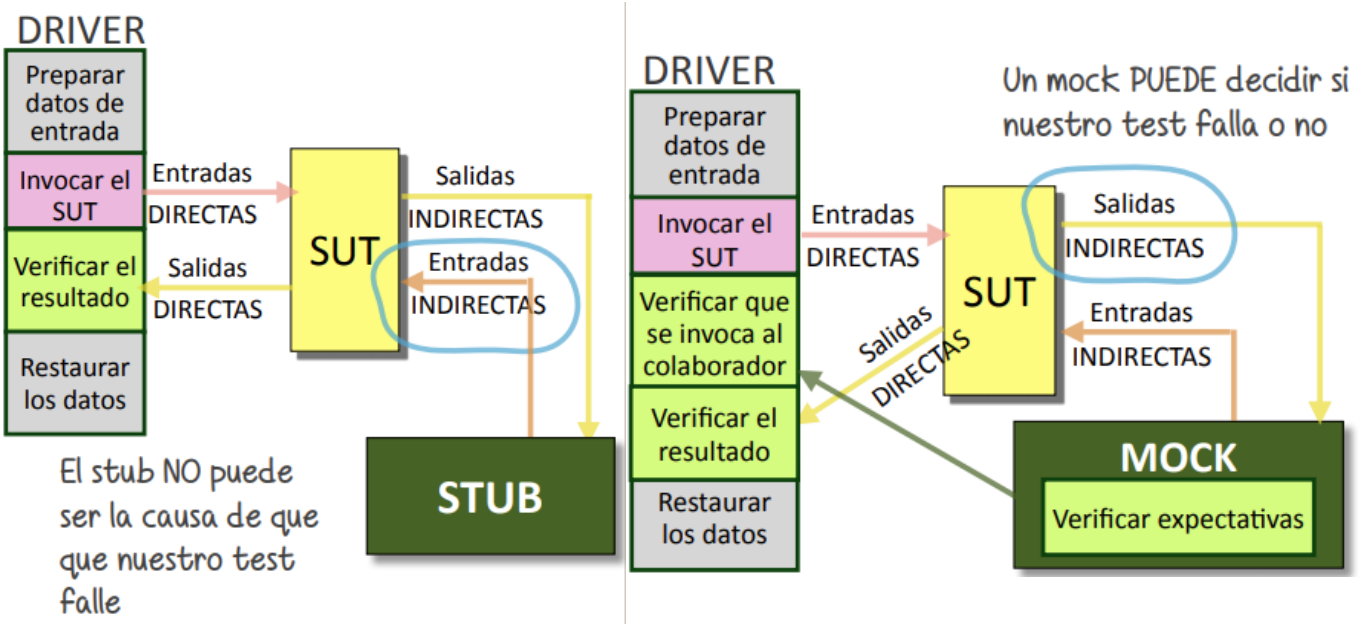
La "implementación" del doble la realizamos "dentro" del test

anyObject() → "cualquier objeto"
anyChar() → "cualquier char"
anyFloat() → "cualquier float"
anyInt() → "cualquier int"

Inyección del doble

Objetos STUB vs Objetos MOCK

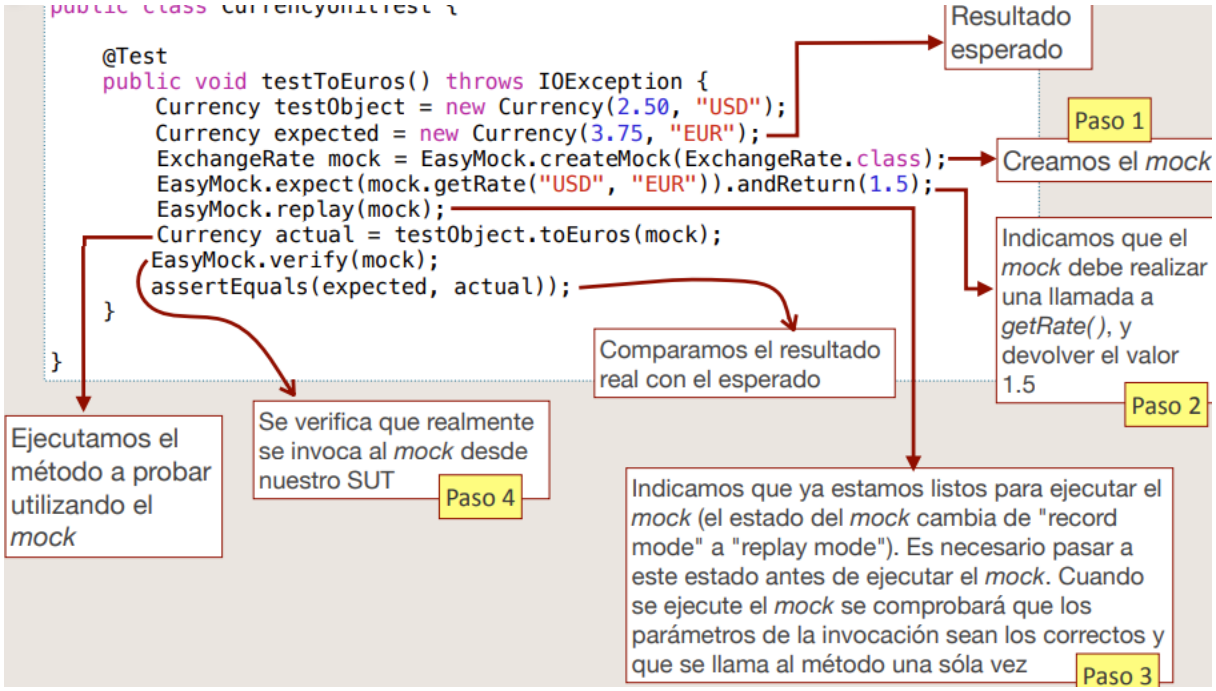
Conceptos STUB vs MOCK



ESTADO vs COMPORTAMIENTO

- Debemos indicar cómo interaccionará nuestro SUT con el objeto mock que hemos creado (**veces invocado, a qué métodos, parámetros, devolución, orden**).
    - Al ejectuar nuestro SUT, el **mock** registrará TODAS las interacciones desde el SUT:
    - Si es un StrictMock y las invocaciones del SUT no coinciden con las expectativas programadas, entonces el doble provocará un fallo (AssertionError).
    - Y si es un Mock, entonces el doble provocará un fallo.
    - En un Mock, el orden de ejecución de las expectativas NO se chequea. En un StrictMock sí, usando un IMocksControl.
    - Si queremos hacer una verificación de comportamiento "estricta" no relajaremos los valores de los parámetros.

Implementación del driver



Partial Mocking

- Implementación ficticia de sólo algunos métodos (partial mocking).
  - Esto ocurre normalmente cuando estamos probando un método que realiza llamadas a otros métodos de su misma clase.
  - Utilizar partialMockBuilder()

```
ToMock mock = partialMockBuilder(ToMock.class)
              .addMockedMethod("mockedMethod").createMock();
```

- Si nuestras dependencias externas no proporcionan entradas indirectas al SUT que debamos controlar, nuestro doble no será un stub, usaremos un mock y por tanto tendremos que utilizar verificación basada en el comportamiento si queremos comprobar que el doble ha sido invocado desde el SUT
- Si los colaboradores proporcionan entradas indirectas al SUT, debemos controlar dichas entradas con un stub para realizar pruebas unitarias. Podemos usar, o no, un framework para implementar los stubs
- Si queremos verificar el comportamiento de nuestro SUT, necesariamente usaremos mocks. Podemos no usar un framework, pero lo habitual es usar alguno

P05

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

VERIFICACIÓN BASADA EN EL ESTADO

- El driver de una prueba unitaria puede realizar una verificación basada en el estado resultante de la ejecución de la unidad a probar.
- Si la unidad a probar tiene dependencias externas, éstas serán sustituidas por **stubs** durante las pruebas. Los **dobles** controlarán las entradas indirectas de nuestro SUT. Un **stub** no puede hacer que nuestro test falle (el resultado del test no depende de la interacción de nuestro SUT con sus dependencias externas)
- Para poder usar los **dobles** (stubs), éstos tienen que poder inyectarse en nuestro SUT a través de los "enabling seam points".

VERIFICACIÓN BASADA EN EL COMPORTAMIENTO

- El driver de una prueba unitaria puede realizar una verificación basada en el comportamiento, de forma que no sólo se tenga en cuenta el **resultado real**, sino también la **interacción de nuestro SUT con sus dependencias externas** (cuántas veces se invocan, con qué parámetros, y en un orden determinado).
- Los dobles usados si realizamos una verificación basada en el comportamiento se denominan **mocks**. Un mock constituye un punto de observación de las salidas indirectas de nuestro SUT, y además registra la interacción del doble con el SUT. Un **mock** sí puede provocar que el test falle.
- Para poder usar los **dobles** (mocks), éstos tienen que poder inyectarse en nuestro SUT a través de los "enabling seam points".
- Para implementar los dobles usaremos la librería **EasyMock**. Para ello tendremos que crear el doble (de tipo Mock, o StrictMock, programar sus expectativas, indicar que el doble ya está listo para ser usado y finalmente verificar la interacción con el SUT.
- **EasyMock** también nos permite implementar drivers usando verificación basada en el estado (usando **stubs**). Para ello tendremos que crear el doble (de tipo NiceMock), programar sus expectativas (relajando los valores de los parámetros) e indicar que el **doble** ya está listo para ser usado por nuestro SUT.