P01A: Entorno de pruebas

IMPORTANTE. A TENER EN CUENTA DURANTE TODO EL CURSO

- Debes subir a Bitbucket las prácticas que realices. **SÓLO SE REVISARÁN** aquellas prácticas subidas **antes** de iniciar la sesión de la siguiente práctica.
- **DURANTE** las clases de **prácticas:**
  - con **carácter general** se darán explicaciones sobre las soluciones de la práctica anterior,
  - con **carácter individual** se realizará el seguimiento del trabajo subido por el alumno (siempre y cuando se haya hecho dentro del plazo establecido) y
  - se resolverán las dudas que surjan.
- Cada alumno tiene asignado un profesor de prácticas, que realizará el seguimiento de vuestro trabajo. No se admitirán cambios de turnos de prácticas que no sean realizados a través de la secretaría del centro.
- Recuerda que tu trabajo de prácticas te permitirá comprender y asimilar los conceptos vistos en las clases de teoría. Y que vamos a evaluar no sólo tus conocimientos teóricos sino que sepas aplicarlos correctamente. Por lo tanto, el **resultado** de tu trabajo **PERSONAL** sobre las clases en aula y en laboratorio determinará si alcanzas o no las competencias teórico-prácticas planteadas a lo largo del cuatrimestre.

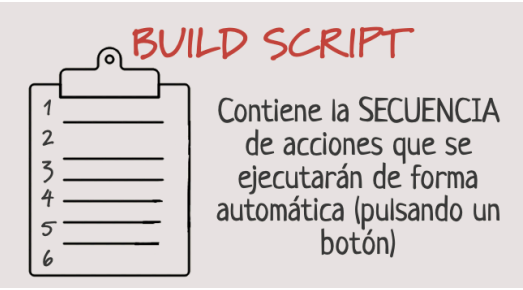
Una vez que tengamos la máquina virtual preparada, nuestro repositorio Git creado y clonado en la máquina virtual, y nuestra licencia Jetbrains activa, ya estamos en disposición de comenzar con nuestro trabajo práctico.

Maven

Maven es una **herramienta de construcción** de proyectos Java. Por definición, la construcción (*build)* de un proyecto es la secuencia de tareas que debemos realizar para, a partir del código fuente, poder usar (ejecutar) nuestra aplicación. Ejemplos de tareas que forman parte del proceso de construcción pueden ser compilación, *linkado*, pruebas, empaquetado, despliegue.... Otros ejemplos de herramientas de construcción de proyectos son *Make* (para lenguaje C), *Ant* y *Graddle* (también para lenguaje Java).

Maven puede utilizarse tanto desde línea de comandos (comando mvn) como desde un IDE.

Cualquier herramienta de construcción de proyectos necesita conocer la secuencia de tareas que debe ejecutar para construir un proyecto. Dicha secuencia de tareas se denomina ***Build Script.***



Maven, a diferencia de Make o Ant (Graddle utiliza elementos de Ant y de Maven), permite definir el *build script* para un proyecto de forma declarativa, es decir, que no tenemos que indicar de forma explícita la "lista de tareas" a realizar, ni "invocar" de forma explícita la ejecución de dichas tareas.

Resumen

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

TEORIA PRACTICA

GIT

- Herramienta de gestión de versiones. Nos permite trabajar con un repositorio local que podemos sincronizar con un repositorio remoto.

MAVEN

- Herramienta automática de construcción 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".

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).

```
1.public void realizaReserva(String login, String password,
2.                           String socio, String [] isbns) throws Exception {
3. ArrayList<String> errores = new ArrayList<String>();
4.    //El método compruebaPermisos() devuelve cierto si la persona que hace
5.    //la reserva es el bibliotecario y falso en caso contrario
6.  if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
7.    errores.add("ERROR de permisos");
8.  } else {
9.    FactoriaBOs fd = FactoriaBOs.getInstance();
10.    //El método getOperacionBO() devuelve un objeto de tipo IOperacionBO
11.    //a partir del cual podemos hacer efectiva la reserva
12.    IOperacionBO io = fd.getOperacionBO();
13.    try {
14.      for(String isbn: isbns) {
15.        try {
16.          //El método reserva() registra la reserva de un libro (para un socio)
17.          //dados el identificador del socio e isbn del libro a reservar
18.          io.reserva(socio, isbn);
19.        } catch (IsbnInvalidoException iie) {
20.          errores.add("ISBN invalido" + ":" +  isbn);
21.        }
22.      }
23.    } catch (SocioInvalidoException sie) {
24.      errores.add("SOCIO invalido");
25.    } catch (JDBCException je) {
26.      errores.add("CONEXION invalida");
27.    }
28.  }
29.  if (errores.size() > 0) {
30.    String mensajeError = "";
31.    for(String error: errores) {
32.      mensajeError += error + "; ";
33.    }
34.    throw new ReservaException(mensajeError);
35.  }
36.}
```

Figura 1. Implementación del método *realizaReserva()*

Resumen

¿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.

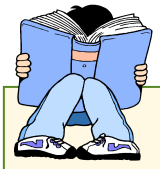
MÉTODO DE DISEÑO DEL CAMINO BÁSICO

- 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.

➪ ➪ ANEXO 2: Observaciones a tener en cuenta sobre la práctica P01B

- Nunca puede haber sentencias en el código que NO estén representadas en algún nodo
- Un nodo solo puede contener sentencias secuenciales, y NUNCA puede contener más de una condición
- El grafo tendrá un único nodo inicio y un único nodo final
- Todas las sentencias de control tienen un inicio y un final, debes representarlos en el grafo.
- Todos los caminos independientes obtenidos tienen que ser posibles de recorrer con algún dato de entrada,
- Cada caso de prueba debe recorrer "exactamente" todos los nodos y aristas de cada camino
- No podemos obtener más casos de prueba que caminos independientes hayamos obtenido
- Todos los datos de entrada deben de ser concretos
- Posiblemente necesitaremos hacer asunciones sobre los datos de entrada (como en la tabla mostrada en el anexo 1).

Resumen



¿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.

de dar de alta, generándose un error con el mensaje: ***"Error en el alta del alumno"***. Si la lista de asignaturas de matriculación es vacía o nula, se genera el mensaje de error: ***"Faltan las asignaturas de matriculación"***. De la misma forma, si para alguna de las asignaturas de la lista ya se ha realizado la matrícula, se devolverá el mensaje de error: ***"El alumno con nif nif_alumno ya está matriculado en la asignatura con código código_asignatura"***.

Nota1: asumimos que todas las asignaturas que pasamos como entrada ya existen en la BD.

Nota2: todos los accesos a la BD se realizan a través de una unidad externa.

El número máximo de asignaturas a matricular será de 5. Si se rebasa este número se devolverá el error ***"El número máximo de asignaturas es cinco"***. El proceso de matriculación sólo se llevará a cabo si no ha habido ningún error en todas las comprobaciones anteriores. Durante el proceso de matricula, puede ocurrir que se produzca algún error de acceso a la base de datos al proceder al dar de alta la matrícula para alguna de las asignaturas. En este caso, para cada una de las asignaturas que no haya podido hacerse efectiva la matrícula se añadirá un mensaje de error en la lista de errores del objeto MatriculaTO. Cada uno de los errores consiste en el mensaje de texto: ***"Error al matricular la asignatura cod_asignatura"*** (siendo cod_asignatura el código de la asignatura correspondiente). El campo asignaturas del objeto MatriculaTO contendrá una lista con las asignaturas de las que se ha matriculado finalmente el alumno.

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes. En el caso del objeto de tipo ***AlumnoTO***, puedes considerar como dato de entrada únicamente el atributo ***nif***. En el caso de los objetos de tipo ***AsignaturaTO*** puedes considerar únicamente el dato de entrada el atributo ***codigo***, que representa el código de la asignatura.

Resumen

¿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 caráter 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.

Resumen

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

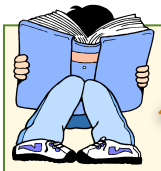
DEPENDENCIAS EXTERNAS

- 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 la 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..

Resumen



¿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".