Resumen

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

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: coordenadas, propiedades, dependencias y plugins. (Hay más secciones, pero de momento sólo veremos estas cuatro)
- Un artefacto maven es un fichero generado por el proceso de construcción de maven y que se identifica 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 para construir un proyecto. Cada ciclo de vida está formado por una secuencia ordenada de fases, cada fase puede tener asociadas unas goals. Una goal siempre pertenece a un plugin. El resultado del proceso de construcción maven puede ser "Build failure" o "Build success".
- El comando "mvn" permite especificar tanto una fase (de alguno de los ciclos de vida) como una goal. En el primer caso se ejecutan todas las goals asociadas a todas las fases del ciclo de vida, desde la primera, hasta la fase especificada. En el segundo caso únicamente ejecutaremos la goal indicada..

TESTS

- Un test está asociado a un caso de prueba, el cual identifica un comportamiento del elemento a probar (cada comportamiento está formado por datos concretos de entrada + resultado concreto esperado)
- 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 en P, siendo S el conjunto de todos los comportamientos especificados y P el conjunto de todos los comportamientos implementados)
- Dependiendo de cómo hayamos diseñado (definido) los casos de prueba, detectaremos más o menos errores (discrepancias entre el resultado esperado y el resultado real).
- Es imposible detectar todos los posibles errores (ya que necesitaríamos un número de casos de prueba impracticable), por lo tanto, nuestras pruebas sólo pueden demostrar la presencia de defectos en el código, pero nunca pueden demostrar la ausencia de ellos. Por lo tanto, como testers, buscaremos detectar el máximo de errores posibles (efectividad) con el menor número posible de casos de prueba (eficiencia)

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.

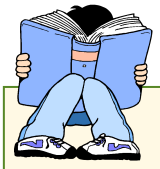
MÉTODO DE DISEÑO DEL CAMINO BÁSICO

- 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, pero nunca podremos darnos cuenta de que nos faltan comportamientos por implementar.
- 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 4: 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 independiente.
- 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 del ejercicio 2)

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)

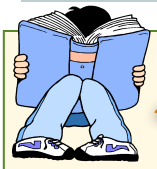
EJECUCIÓN DE LOS TESTS

- 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 para integrar la automatización de las pruebas en el proceso de construcción del proyecto y qué ficheros genera nuestro proceso de construcción en cada caso.

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.

MÉTODO DE PARTICIONES EQUIVALENTES

- Necesitamos identificar previamente todas 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, (si la validez de una partición de entrada, depende de otra/s de las entradas, entonces hay qua agruparlas y particionar todas ellas a la vez). 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 en cada caso de prueba (sólo puede haber una partición de entrada inválida 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 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 (DOCs). 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 DOC, 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"

IMPLEMENTACIÓN 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..