P10- Análisis de pruebas: Cobertura

Cobertura

Un análisis de la cobertura de nuestras pruebas nos permite conocer la **extensión** de las mismas. Aunque se trata de una métrica que no mide la efectividad de nuestras pruebas, es evidente que si un código no está siendo probado, no podemos ser efectivos. En esta sesión utilizaremos la herramienta JaCoCo para analizar tanto la cobertura de nuestros tests de forma automática.

Esta herramienta se integra con Maven, de forma que podemos incluir el análisis de cobertura en la construcción de nuestro proyecto.

GitHub

El trabajo de esta sesión también debes subirlo a *GitHub*. Todo el trabajo de esta práctica deberá estar en el directorio **P10-Cobertura** dentro de tu espacio de trabajo.

➪ ➪ Ejercicio 1: Proyecto cobertura

Crea un proyecto Maven, usando los siguientes datos:

- ***Name***: ***cobertura***
- ***Location***: ***$HOME/ruta_de_tu_directorio_de_trabajo/P10-Cobertura/***
- ***JDK***: nos aseguraremos de que esté seleccionado JDK 21
- ***groupId: ppss.P10*** ; ***ArtifactId: cobertura***

Una vez creado el proyecto recuerda que debes BORRAR **SIEMPRE** el fichero ***.gitignore*** que ha generado IntelliJ.

Crea la carpeta ***intellij-configurations*** en la raíz de tu proyecto en donde guardaremos las *run-configurations* asociadas.

En el directorio ***Plantillas-P10/ejercicio1*** tienes los ficheros java que usaremos en este ejercicio. El fichero Sample.java contiene nuestro código a probar. Añade el fichero en el paquete ***ppss.ejercicio1.*** El resto de ficheros de la carpeta *ejercicio1* contienen tests unitarios, tests de integración, y los dobles correspondientes para realizar las pruebas unitarias. Añádelos donde corresponda y **familiarízate con el código proporcionado**.

A) Lo primero que haremos será modificar el **pom** convenientemente para:

- automatizar los tests con *junit* (puedes copiar la dependencia de proyectos anteriores)
- compilar el proyecto con la versión 3.13.0 del compilador java
- ejecutar los tests unitarios usando la versión 3.5.2 del plugin correspondiente

> **Importante:** Recuerda que los plugins ***compiler*** y ***surefire*** ya están incluidos por defecto en el pom, pero los añadimos siempre porque queremos usar la versión 3.5.2, que es posterior a la que se incluye por defecto.

- ejecutar los tests de integración usando la versión 3.5.2 del plugin correspondiente.
- instrumentar nuestros tests con JaCoCo (tanto los unitarios como los de integración), y también vamos a obtener los informes correspondientes en formato html, por lo que tendrás que añadir el plugin de JaCoCo con las goals que corresponda.

Importante: Las goals ***report*** y ***report-integration*** del plugin JaCoCo están asociadas por defecto a la fase verify, pero NO son las únicas. La goal failsafe:verify también está asociada a la misma fase.
Si hay varias goals asociadas a la MISMA fase de Maven, éstas se ejecutan en el orden en el que aparecen en el pom. Por ejemplo, si incluyes la goal ***failsafe:verify*** antes de la goal ***jacoco: integration-report***, entonces, si algún test de integración falla NO generaremos el informe de cobertura para los tests de integración, ya que la goal f***ailsafe:verify*** detendrá el proceso de construcción.
Si escribimos la goal ***jacoco:report-integration*** antes que la goal ***failsafe:verify,*** entonces generaremos primero el informe de cobertura y posteriormente comprobaremos si algún test de integración ha fallado y se detendrá el proceso de construcción.
Obviamente NO estamos interesados en generar ningún informe si nuestros tests fallan por lo que asegúrate de que en el pom aparece PRIMERO la goal ***failsafe:verify,*** y posteriormente la goal ***jacoco:report-integration***. En este caso, el ORDEN entre ellas IMPORTA!!!

Importante: Si se incluyen las goals ***prepare-agent*** y ***prepare-agent-integration***-del plugin JaCoCo (las DOS), entonces se calculará la cobertura de los tests unitarios y los de integración POR SEPARADO (la cobertura de los tests unitarios se guarda por defecto en el fichero ***jacoco.exec*** y la cobertura de los tests de integración en el fichero ***jacoco-it.exec***).
Todos los cálculos se realizan en memoria cuando se ejecutan los tests correspondientes.
Si no incluimos la goal *prepare-agent-integration*, todos los cálculos de TODOS los tests se incluirán en el fichero ***jacoco.exec*** (independientemente de que se trate de un test unitario o de un test de integración

B) Crea una ***Run Configuration*** con el nombre ***run_Unit_tests_Sample.maxValue*** para ejecutar los tests unitarios del método Sample.maxValue() y para generar el informe de cobertura correspondiente.

Para ejecutar las pruebas unitarias sólo del método *Sample.maxMaxvalue()* debes usar la propiedad `test` (del plugin *surefire)* que ya conoces de prácticas anteriores (dicha propiedad debe ir precedida por `-D`, es decir: `-Dtest=`*nombre_clase_a_probar*).

Dado que la generación del informe está asociada a la fase ***verify*** tendremos que ejecutar dicha fase y excluir la ejecución de los tests de integración (que también están asociados a la misma fase). En este caso usamos la propiedad `skipITs` (del plugin *failsafe)*.

También estamos interesados en eliminar cualquier fichero generado por Maven de construcciones previas (si no lo hacemos así, probablemente obtendremos informes de cobertura incorrectos, ya que JaCoCo no borra los valores de los contadores de construcciones previas.

En definitiva, necesitamos usar el siguiente comando maven:

```
mvn clean verify -Dtest=Test_Sample_maxValue -DskipITs=true
```

C) Observa en la consola la salida de maven y **comprueba qué goals se ejecutan y en qué orden** cuando construimos el proyecto con la run configuration ***run_Unit_tests_Sample.maxValue***

D) Visualiza en el navegador el informe cobertura generado por JaCoCo. Puedes abrir directamente el fichero ***index.html*** del informe en el navegador desde el menú contextual, seleccionando ***Open in → Browser***

- ¿Cuántos y qué informes de cobertura se han generado teniendo en cuenta que en el pom hemos indicado que se deben generar dos tipos de informes?

Consulta el **informe generado** y contesta a las siguientes cuestiones:

- ¿Cual es el número de instrucciones *bytecode* ejecutadas para la clase *Sample*?
- ¿Cuántas instrucciones *bytecode* nos faltan para tener una cobertura del 100% de instrucciones *bytecode* para la clase *Sample*?
- ¿Cuántas instrucciones bytecode tiene la clase *Sample*?

- En el informe verás el método Sample(), sin embargo no lo hemos implementado en el código que se os ha proporcionado. ¿Por qué aparece entonces en el informe?
- El método Sample() tiene una cobertura de branches con valor "n/a". ("n/a" significa "not applicable"). Razona por qué tiene dicho valor.
- Indica cuál es el **valor de complejidad ciclomática** para el método Sample.maxValue() y justifica, teniendo en cuenta el código del método y la fórmula que usa JaCoCo, por qué tiene ese valor, (debes indicar qué líneas de código se asocian a los valores calculados por la fórmula de JaCoCo.
- Indica **qué representa** el valor de **Cxty** para *Sample.maxValue()*, y razona si según ese valor estamos siendo eficientes y efectivos (no debes tener en cuenta que Cxty es una cota máxima).
- Observa cuáles son los valores de cobertura para *Sample.isValid()*. Teniendo en cuenta que el código de *maxValue()* contiene la sentencia"*if (isValid(data))*" razona por qué en el informe de cobertura no se recorre esa línea.

E) Crea una ***Run Configuration*** con el nombre ***run_Unit_tests_Only*** para ejecutar solamente los tests unitarios y generar el informe de cobertura correspondiente(debes omitir la ejecución de los tests unitarios igual que hemos hecho en la *run configuration* anterior).

Construye el proyecto y visualiza el informe de cobertura. ¿Cuál es la cobertura de condiciones+decisiones a nivel de proyecto? Haz lo necesario para conseguir un 100% de cobertura (a nivel de proyecto) de decisiones + condiciones. Viendo el informe, ¿cuántos tests como máximo te harán falta para eso?

F) Crea una ***Run Configuration*** con el nombre ***run_Integration_tests_only***_Only para ejecutar solamente los tests de integración y generar el informe correspondiente. En este caso, para omitir los tests de integración usa la propiedad `surefire.excludes`. Puedes usar una expresión regular como:`**/Test*` para excluir la ejecución de todos los tests unitarios

Construye el proyecto y visualiza el informe de cobertura. Razona por qué todos los valores de cobertura para el método countValues es cero.

Añade un **nuevo tests de integración** para maxValue(), en concreto usa como entrada un array con valor null. El resultado esperado debe ser null.

Ejecuta de nuevo los tests de integración. Verás que se detecta un error.

Puedes comprobar que la causa del error es una de las que comentamos en clase de teoría (malentendido sobre la interfaz), ya que al hacer las pruebas unitarias sobre *isValid()* se ha asumido que nunca vamos a tener como entrada un valor null.

Depura el error y ejecuta la fase verify desde la ventana de maven (fíjate que en este caso vas a ejecutar primero los tests unitarios y a continuación los tests de integración, y vas a obtener dos informes de cobertura asociados a cada uno de ellos.

¿Por qué ahora los tests unitarios no generan un informe con un 100% de cobertura de *branches*?

➪ ➪ Ejercicio 2: Informes de cobertura

En el proyecto anterior, vamos a crear un nuevo paquete ***ppss.ejercicio2***, en el que deberás copiar las clases de la carpeta /plantillas-P10/ejercicio2. Cada fichero debes añadirlo donde corresponda.

A) Crea una nueva ***Run Configuration*** con el nombre ***run_all_tests*** (recuerda que debes borrar el directorio target previamente), para ejecutar todos los tests, tanto los unitarios como los de integración, y obtener los informes de cobertura por separado de los tests unitarios y de los de integración. Construye el proyecto y comprueba que se generan los informes.

Con respecto al informe de **tests unitarios**:

- Justifica el valor de cobertura de *branches* para la clase *ppss.ejercicio2.MyClass*.
- Explica por qué el valor de cobertura de branches para la clase *ppss.ejercicio2.Lock* es del 100%, a pesar de que no todos sus métodos tienen también el valor 100%.
- Observa también que para la clase *ppss.ejercicio2.Lock* la cobertura de líneas es del 72,7% mientras que la cobertura de *branches* es del 100% ¿cómo es eso posible?

B) Queremos "**chequear**" (goal check) que se alcanzan ciertos niveles de cobertura con nuestras **pruebas unitarias**. Modifica el pom para que se compruebe de forma automática las siguientes dos "reglas" (<rules>) :

- ➡ A nivel de **proyecto**, queremos conseguir una CC con un valor mínimo del 82%, una cobertura de instrucciones mínima del 90%, y que no haya ninguna clase sin probar.
- ➡ A nivel de **clase**, queremos conseguir como mínimo una cobertura de líneas del 75% y solo nos puede quedar el 40% de métodos sin probar como mucho.

C) Obtén de nuevo los informes de cobertura, y comprueba que el proceso de construcción falla. porque se incumplen las dos reglas (<rule>) que hemos impuesto.

Indica qué casos de prueba deberías añadir para obtener una construcción exitosa.

➪ ➪ Ejercicio 3: Proyecto Matriculacion

Para este ejercicio usaremos el proyecto multimódulo ***matriculacion*** de la práctica P06B.

Para ello copia tu solución, es decir, la carpeta *matriculacion* (y todo su contenido) en el directorio de esta práctica *P10-Cobertura/*.

Una vez copiada la carpeta *matriculacion*, simplemente abre el proyecto ***matriculacion*** desde IntelliJ (seleccionando la carpeta que acabas de copiar).

Se pide:

A) Modifica convenientemente el pom del módulo ***matriculacion-dao*** para obtener un informe de cobertura para dicho módulo (tanto para los tests unitarios como para los tests de integración). Genera el informe a través del correspondiente comando maven (puedes obtener los informes directamente desde este módulo).

B) Observa los valores obtenidos a nivel de proyecto y paquete. Tienes que **tener claro cómo se obtienen** dichos valores. Fíjate también en los valores de CC obtenidos a nivel de paquete y clase.

C) El proyecto *matriculacion-dao* también ejecuta las clases de *matriculacion-comun*, sin embargo no aparecen en el informe (es obvio porque los ejecutables de *matriculacion.comun* no están instrumentados).

D) Dado que tenemos un proyecto multimódulo, vamos a usar la goal *jacoco:report-aggregate* para generar un informe para las dependencias de cada módulo, a partir del mecanismo reactor de maven. (ver https://www.jacoco.org/jacoco/trunk/doc/report-aggregate-mojo.html)

Para ello tendrás que comentar el plugin *jacoco* del proyecto *matriculacion-dao* y configurar el pom del proyecto *matriculacion* para obtener los informes unitarios, de integración y además el informe agregado (goal *report-aggregate)*. Si observas la documentación del enlace del apartado anterior, verás que dicha goal no está asociada por defecto a ninguna fase de maven. Debes tener claro qué ocurrirá si no la asociamos a ninguna fase, así como a qué fase deberemos asociarla.

E) Ejecuta el comando maven correspondiente para obtener el informe agregado de cobertura para el proyecto multimódulo (así como los informes de cobertura para los tests unitarios y de integración para todos los módulos del proyecto). Averigua dónde se genera el informe agregado de cobertura. Observa las diferencias con el informe que hemos obtenido anteriormente.

Resumen

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

NIVELES DE COBERTURA

- La cobertura es una métrica que mide la extensión de nuestras pruebas. Existen diferentes variantes de esta métrica, que se pueden clasificar por niveles, de menos a más cobertura. Es importante entender cada uno de los niveles.
- El cálculo de esta métrica forma parte del análisis de pruebas, que se realiza después de su ejecución..

HERRAMIENTA JaCoCo

- JaCoCo es una herramienta que permite analizar la cobertura de nuestras pruebas, calculando los valores de varios "contadores" (JaCoCo counters), como son: líneas de código, instrucciones, complejidad ciclomática, módulos, clases,... También se pueden calcular los valores a nivel de proyecto,, paquete, clases y métodos.
- JaCoCo puede usarse integrado con maven a través del plugin correspondiente. Es posible realizar una instrumentación de las clases on-the-fly, o de forma off-line. En nuestro caso usaremos la primera de las opciones.
- JaCoCo genera informes de cobertura tanto para los tests unitarios como para los tests de integración. Y en cualquier caso, se pueden establecer diferentes "reglas" para establecer diferentes niveles de cobertura dependiendo de los valores de los contadores, de forma que si no se cumplen las restricciones especificadas, el proceso de construcción no terminará con éxito.
- De igual forma, para proyectos multimódulo, se pueden generar informes "agregados", de forma que el informe agregado de cada módulo "incluye" los informes de cobertura de los módulos de los que depende.

Observaciones sobre esta práctica

EJERCICIO 1:

- En este primer ejercicio tenéis que familiarizaros con los informes de cobertura de JaCoCo y tenéis que tener claros qué significan los valores de TODOS contadores (considerando todas las columnas), y, teniendo en cuenta que dichos informes se proporcionan en varios niveles.
- También hay que tener claro cómo hay que configurar el pom para poder generar los informes de cobertura de los tests unitarios.
- Si no necesitamos ver el informe en formato html NO es necesario incluir las goals correspondientes (*jacoco:report y jacoco:report-aggregate*). Dichas goals NO calculan ninguna métrica. Los cálculos de cobertura se llevan a cabo en memoria cuando se ejecutan los tests
- Si incluimos en el pom varias goals asociadas a la misma fase, éstas se ejecutan en el ORDEN en el que aparecen en el pom cuando se alcance dichas fase
- Si sólo incluimos la goal *jacoco:agent* el fichero jacoco.exec contendrá la cobertura de cualquier test ejecutado, con independencia de si es un test unitario o un test de integración.
- Es IMPORTANTE no sólo analizar los resultados sino tener claro CÓMO configurar el pom y qué COMANDO maven necesitamos ejecutar para poder obtener la cobertura de los tests y los informes, en formato html (en caso de ser necesarios)

EJERCICIO 2:

- Tenéis que saber modificar correctamente el pom, para configurar la goal ***jacoco:check*** para poder incluir las comprobaciones necesarias en el proceso de construcción y garantizar que nuestro código satisface ciertos niveles de cobertura..
- Podemos configurar la goal jacoco:check con cualquier número de reglas. A su vez, cada regla puede imponer restricciones sobre cualquiera de las seis métricas que calcula JaCoCo..

EJERCICIO 3:

- El ejercicio 3 usa el proyecto multimódulo matriculación, y debe ser un proyecto IntelliJ totalmente independiente del proyecto "cobertura" de los ejercicios anteriores. Es decir, para esta práctica tendremos dos proyectos IntelliJ: uno con un proyecto maven ("cobertura") y otro con un proyecto maven multimódulo ("matriculacion").
- En este ejercicio obtenemos 3 informes de cobertura diferentes: para los tests unitarios, para los tests de integración y los informes agregados. Hay que tener claro dónde (en qué carpeta) se genera cada uno de ellos. Así como dónde y en qué ficheros se almacena el resultado del análisis de cobertura cuando se ejecutan los correspondientes tests.
- Recuerda que el proyecto matriculación es un proyecto maven multimódulo. Usamos la relación de herencia para evitar en lo posible redundancias en las configuraciones del pom de los diferentes módulos.
- Observa los diferentes informes para tener claro como calcula JaCoCo las diferentes métricas en cada uno de los niveles.