Análisis de código estático

El análisis de código estático es una técnica de revisión de código que examina el código fuente sin ejecutar el programa. Se utiliza para identificar errores, vulnerabilidades de seguridad y malas prácticas de codificación. Para el análisis del proyecto base del obligatorio de la materia Ingeniería de Software Ágil 2, decidimos utilizar la herramienta SonarQube cuyo reporte completo puede ser encontrado en la carpeta de Anexos (ver carpetas Back y Front –dentro de carpeta Anexos– con los respectivos documentos).

Reporte SonarQube Back

En el *backend*, las clasificaciones automáticas generadas por la herramienta fueron las siguientes: *Reliability B, Security A, Security Review E* y *Maintainability* A; es decir, el proyecto no tiene grandes problemas –sobre todo de deuda técnica– que deban ser urgentemente solucionados para lograr un código más mantenible. Sin embargo, fueron detectados algunos puntos débiles que pasaremos a mencionar y analizar brevemente.

Fueron detectados 104 *code smells* –documento: *Back CodeSmells*– dentro de los cuales existen dos tipos que nos gustaría mencionar:

- inicialización de variables no utilizadas; y,
- lanzamiento de excepciones generales

En el caso del primer *smell*, implica el desperdicio de recursos –sobre todo de memoria–, mientras que el segundo sí entendemos que es una práctica de programación que deba ser mejorada.

Referente a la seguridad –documento: *Back Security HotSpots*–, podemos afirmar que –dada su clasificación– representa el punto más débil del código. *SonarQube* identifica dos grandes puntos de mejora:

- criptografía débil; y,
- configuración insegura

El primer punto de mejora, entendemos que aplica para un contexto de producción en el cual la información debe ser tratada en concordancia con determinados estándares que, en este caso –y por tratarse de un trabajo académico–, es prescindible. A su vez, el segundo punto de mejora también tiene una característica asociada a lo esperable para un código comercial, pero en este caso la seguridad asociada al permitir cualquier origen no tiene tanta relevancia.

Por último, el reporte presenta 9 *bugs* –documento: *Back Bugs*– en el *backend* siendo todos estos por no sobreescribir el método *GetHashCode* en clases que sobreescriben el método *Equals.*

De acuerdo con la estimación de *SonarQube*, corregir la deuda técnica del *backend* por completo llevaría aproximadamente 16 horas y 30 minutos.

Reporte SonarQube Front

En el *frontend*, todas las clasificaciones automáticas generadas por la herramienta son clase A, excepto por *Reliability* que tuvo clasificación C; es decir, el proyecto tampoco tiene grandes problemas –sobre todo de deuda técnica– que deban ser urgentemente solucionados para lograr un código más mantenible. Sin embargo, la presencia de *bugs* implica una peor clasificación de *Reliability* y debe ser analizado para saber a qué se debe particularmente.

El reporte indica la presencia de 71 *code smells* y 32 *bugs.* Dentro de los primeros (*code smells,* documento: *Front CodeSmells*), la gran mayoría se debe a atributos obsoletos (*deprecated*); mientras que los segundos (*bugs,* documento: *Front Bugs)* refieren a agregar una descripción a un objeto (tabla, *tag*, *fieldset*, etc). Ninguno de estos elementos representa una dificultad significativa, pero sí son la suficiente cantidad como para considerar si su resolución es necesaria en el alcance de este trabajo.

De acuerdo a lo estimado en el reporte, la corrección de los 71 *code smells* llevaría aproximadamente 5 horas y 39 minutos.

Se incluye tanto para el *frontend* como para el *backend* un documento –*Front Overview* y *Back Overview,* respectivamente– para visualizar el resumen del análisis de *SonarQube.*

Análisis de la solución

Entendemos que la granularidad de la solución puede ser mejorada en el sentido que existen paquetes que son demasiado extensos y podrían ser divididos en otros más pequeños. Un claro ejemplo de esto es el tratamiento de la lógica del dominio en un único paquete en vez de utilizar un paquete con la lógica asociada a cada uno de los elementos. Esto permitiría no solamente comprender de manera más simple la solución, sino que permitiría, a través de la abstracción de servicios para interfaces, contribuir a disminuir el acoplamiento, mejorar la cohesión de cada paquete y permitir que el código esté alineado con OCP; es decir, generar código que pueda ser modificado (por ejemplo cambiar la lógica aplicada a la venta de tickets) desarrollando una nueva clase que introduzca los cambios necesarios implementando la interfaz.

Lo anterior permitiría que cada uno de los paquetes con la lógica del dominio posea una interfaz que exponga los servicios, y que todo el resto de la implementación sea solamente accesible dentro del propio paquete. Esto mejoraría significativamente algunas métricas obtenidas en el análisis con NDepend disponible en la documentación.