Este es el post del grupo ES-106 para el trabajo en grupo de los seminarios de la asignatura ASW. El grupo está formado por Adriana Herrero González, Daniel González Pérez y Miguel Morís Gómez

¿Qué es el mutation testing?

Hoy en día existen muchos tipos de test (unitarios, de integración, de usabilidad…) y también diferentes maneras de implementarlos (implementar todo el código y después testearlo, usar otras técnicas como TDD) en función de la metodología de desarrollo.

Antes de responder a la pregunta de definir que es el mutation testing deberíamos de plantearnos primero por qué hacemos test de nuestras aplicaciones.

La respuesta más natural es porque queremos asegurarnos de que nuestra aplicación funcione correctamente y satisfaga los requisitos del cliente pero, una vez disponemos de un código y unos test que pasan correctamente, ¿significa esto que el código es correcto? La respuesta es que depende de si los test están planteados correctamente pero… ¿cómo podemos saber si los test testean bien nuestro código? ¿Acaso deberíamos de hacer test de los test?

Mutation testing es en definitiva una técnica que nos permite evaluar la fiabilidad de los test que estamos realizando.

Consiste en simular bugs deliberadamente y ver cuántos de nuestros test fallan.

La motivación detrás de esta manera de proceder pasa por entender que la programación no deja de ser un proceso iterativo de prueba y error en el que con toda seguridad y en tanto que somos humanos, vamos a introducir bugs siempre, y todavía más cuando el código que estamos modificando ni siquiera es nuestro, algo muy común hoy en día.

La “gracia” del mutation testing es introducir “mutaciones” que no son evidentes que sean bugs, que son precisamente los errores difíciles de solucionar.

Goran Petrovic defiende que el mutation testing es una herramienta más fiable que la cobertura de código.

En pocas palabras, las herramientas de cobertura de código cuentan el porcentaje de líneas de código que están cubiertas por test y te ofrecen un porcentaje, cuanto más alto sea ese porcentaje mejor.

Con las mutaciones el procedimiento es generar todas las mutaciones posibles en el código, ejecutar la batería de test, y calcular el porcentaje dividiendo las “bajas” (test que han fallado) entre el número total de test.

Una vez hecho esto, deberemos de fijarnos en aquellos test que aún siguen pasando para intentar mejorarlos y hacerlos más rigurosos.

Como desarrolladores no tenemos que preocuparnos por generar mutaciones porque hay una gran variedad de herramientas para mutation testing automatizado que hacen esto por nosotros.

Pruebas de Mutación para mejorar el diseño

Las pruebas de mutación no solo robustecen los tests unitarios, sino que también revelan oportunidades para refactorizar y simplificar el diseño del código. Esto resulta en un código más mantenible y claro, algo difícil de lograr solo con pruebas convencionales.

Mutation Test VS Mutation Analysis

El análisis de mutación mide la efectividad de las pruebas a través de métricas como el número de mutantes "vivos" (detectados), el ratio de vivos/totales, etc. Las pruebas de mutación, por otro lado, aplican estas métricas para mejorar el código y encontrar errores difíciles de detectar.

Herramientas de Mutation Testing en Google

Las pruebas de mutación se realizan en la fase de revisión del código. Después de hacer una pull request, se produce el análisis de mutación y se calculan parámetros relevantes. Disponiendo de estas métricas y heurísticos, y basándose en ellas, se generan las mutaciones apropiadas. Con estas mutaciones, los tests se pasan nuevamente y es generado un informe para el desarrollador, donde se sugieren cambios y mejoras en base a los mutantes que viven.

Los lenguajes implementados en servidores de mutación son: C++, Python, Java, Go, TypeScript, JavaScript, Kotlin, Dart, Common Lisp, SQL y Rust.

Las implementaciones de estos mutadores o servidores de mutación se sustentan sobre conceptos más técnicos como teoría de compiladores, AST's, etc.

Aunque en la práctica la implementación es más sencilla, la idea inicial es mutar un AST completo, con todas las variaciones en todas las partes posibles.

Esto, por ser costoso computacionalmente y suponer una carga de trabajo excesiva para el desarrollador, no se hace, y se mutan solo algunas partes del código, siendo el proceso previo de análisis y filtrado muy importante.

Como referencia, solo se introducen 5-10 mutaciones por cada 100 línea, y nunca más de una por línea. Tampoco se trabaja con lenguaje intermedio como Bytecode para no complicar la labor del programador. De igual forma, y a diferencia de

algunas implementaciones open source, no se usan "megamutaciones", aunque están abiertos a dicha posibilidad.

Papel de los *mutantes* en las pruebas del software

Goran menciona en la entrevista que, él mismo junto a un grupo de ingenieros de Google, realizaron estudios que identificaron hasta 15 millones de mutaciones en el código de varios proyectos de la empresa, en un periodo que duró 6 años. El entrevistador le pregunta si mostrar esas mutaciones a los desarrolladores los animaba o no a escribir más casos de prueba. La respuesta es sí. Para reforzar su argumento, utiliza ejemplo de las pull request. Una pull request es una solicitud que contiene una cierta cantidad de código y una cierta cantidad de pruebas, enviada a un compañero para su revisión. Si se envía directamente, sin revisión y sin proponer mutaciones, su contenido probablemente permanecerá igual al momento de ser aprobada. Sin embargo, cuando se incluyen comentarios de pruebas de mutación en la revisión, como "por favor, añade una prueba para eliminar este mutante", se observa que el código final enviado contiene más pruebas en comparación con los casos en los que no se introducen mutaciones. Además, Goran explica que no solo aumenta la cantidad de pruebas, sino también su calidad.

En la entrevista, Goran explica el concepto de **efecto de acoplamiento**, que se basa en la hipótesis de que los test cases capaces de detectar pequeños cambios artificiales en el código, también son efectivos para identificar errores reales más complejos. Para respaldar esta idea, Goran menciona un estudio que realizó en Google, donde analizaron cientos o incluso miles de errores reales en el código. Descubrieron que, en el 69% de los casos, si se hubiera utilizado *mutation testing*, se habrían podido detectar y prevenir muchos de estos errores antes de que llegaran a producción.

Goran Petrovic cuenta cómo llegó a trabajar con ***mutation testing* en Google.** Inicialmente, no tenía experiencia con el tema, pero heredó un proyecto de otros después de un hackathon y, con el tiempo, lo convirtió en su trabajo principal.Explica que para que un proyecto de este tipo sea viable, hay que adaptarlo a las limitaciones existentes en lugar de intentar implementar una versión idealizada. Llama a esta estrategia *desarrollo impulsado por la capacidad*, donde se prioriza lo que es factible en lugar de buscar la perfección.

Se han desarrollado **heurísticos** y técnicas para mejorar el mutation testing, destacando la retroalimentación de usuarios como clave. Existen cinco o seis heurísticos con múltiples instancias, y se usan ejemplos como colecciones de Java para ilustrarlos. También se han identificado estrategias para descartar mutaciones inútiles, priorizando aquellas que afectan la lógica de negocio.