Introdução:

Mutation testing é um tipo de teste de software no qual certos aspetos do código-fonte são alterados / mudados para verificar se os casos de teste são capazes de encontrar erros no código-fonte. O objetivo do mesmo é garantir a qualidade dos casos de teste em termos de robustez para que este falhe no código-fonte alterado.

As alterações feitas no programa mutante devem ser mantidas extremamente pequenas para não afetar o objetivo geral do programa, assim como não aumentar de forma exponencial o tempo de compilação dos mesmos.

Processo:

1. Usa-se um programa e um conjunto de testes previamente gerados para esse programa. Esses testes podem ter sido desenvolvidos com várias técnicas, como Category Partition, Static Testing, entre outros...
2. É criado um conjunto de programas semelhantes (mutantes), cada um diferente do original, num determinado aspeto;
a. Por exemplo, substituindo um operador de adição por multiplicação
3. Os dados do teste original são então executados nos mutantes
4. Se os testes detectarem diferenças em mutantes, então os mutantes são considerados mortos, e o conjunto de teste é considerado adequado.

Um mutante permanece vivo se:

- é equivalente ao programa original (funcionalmente idêntico embora sintaticamente diferente - chamado de mutante equivalente) ou,
- o conjunto de teste é inadequado para matar o mutante

No último caso, os casos de teste precisam ser aumentados (adicionando um ou mais novos casos de teste) para matar o mutante
Para a geração automática de mutantes, são usados operation mutators, que é um programa predefinido regras de modificação (ou seja, correspondendo a um modelo de falha), entre os quais:

- **AOR** - Substituição do Operador Aritmético
    - Substitui um operador aritmético por outro operador aritmético. Os operadores aritméticos são +, -, *, /,%.
- **ROR** - Substituição de Operador Relacional
    - Substitui um operador relacional por outro operador relacional. Os operadores relacionais são <=,> =,! =, ==,>, <.
- **COR** - Substituição de operador condicional
    - substitui um operador condicional por outro operador condicional. Os operadores condicionais são &&, ||, &, |,!, ^.

- **AOR** - Substituição do Operador de Atribuição
    - Substitui uma atribuição de operador por outro operador de atribuição. Operadores de atribuição incluem =, + =, - =, / =.
- **SVR** - Substituição de Variável Escalar
    - Substitui cada referência de variável por outra referência de variável que foi declarada no código.

Mutation score dos testes realizados em entregas anteriores

Excluindo os packages relacionados com a GUI, resultaram 5 packages.

- com.chschmid.jdotxt
    - Cuja mutation coverage é de 18%
- com.chschmid.jdotxt.util
    - Cuja mutation coverage é de 5%
- com.todotxt.todotxttouch.task
    - Cuja mutation coverage é de 12%
- com.todotxt.todotxttouch.task.sorter
    - Cuja mutation coverage é de 0%
- com.todotxt.todotxttouch.util
    - Cuja mutation coverage é de 52%

O resultado das 38 classes, distribuídas por estas 5 packages, corresponde a uma mutation score de 21%.

É um valor francamente baixo, indicativo da volatilidade do código-fonte e quanto está exposto a bugs.

Equivalência de mutantes

Por vezes, fazer uma alteração no código-fonte não é suficiente para mudar o comportamento do mesmo. O código alterado é logicamente equivalente ao código original. Nesses casos, não é possível escrever um teste que irá falhar para o mutante, tendo em conta que também não irá falhar para o código não mutado. Infelizmente, é impossível determinar automaticamente se um mutante sobrevivente é um mutante equivalente ou apenas carece de um caso de teste eficaz. Esta situação requer um ser humano para examinar o código. E isso pode levar algum tempo. De acordo com alguns estudos, demora 15 minutos, em média, para avaliar se um mutante é equivalente.

Problemas ocorridos:

- Grande parte das classes que contém bastantes mutantes tem a notação “private”, indicando que o seu acesso é feito a partir da própria classe.
- Também outros métodos contém parâmetros que não são possíveis de testar, tais como InputStream, OutputStream, File, dificultando assim a testagem

- Assim, não é possível testar esses métodos e, consequentemente, matar os mutantes que lá se encontram.

Descrição dos testes usados para aumentar a mutation score

Para fazer os testes, decidimos procurar no código-fonte do projeto branches que pudessem ser exploradas. Visto que o objetivo de mutation-testing será alterar o código-fonte de modo a provar os testes fidedignos, decidimos aumentar a cobertura geral.

Para isto, começámos por alterar testes antigos e criar novos testes para atingir o máximo de branches possíveis, ao fazer isto, começou a aumentar a mutational coverage nos relatórios.

A principal classe testada foi a classe Task. Uma classe não abrangida anteriormente, e com muita utilização ao longo do projeto. Ao adicionar testes ao máximo de branches aumentou a cobertura geral, devido a ser possível ver casos limite a serem quebrados após alterações ao código fonte.