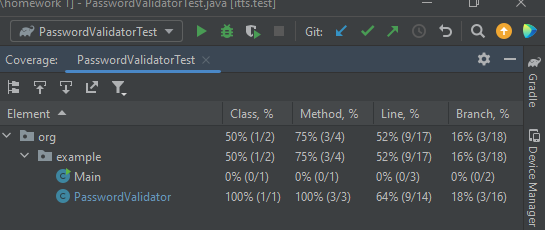
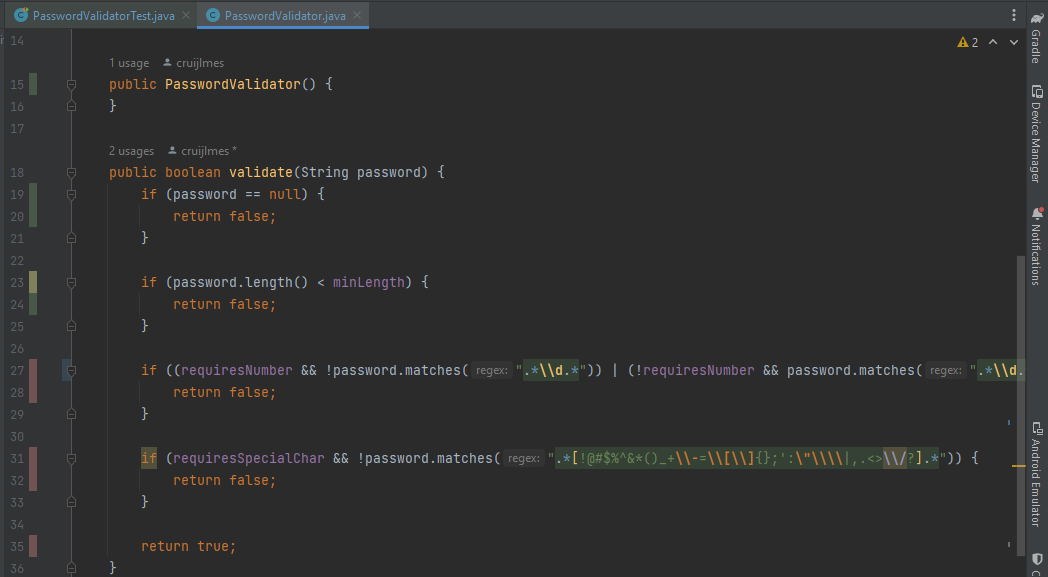
HOMEWORK 2

Task1

In questa task dell’homework 2 abbiamo completato la suite di test black-box con gli structural testing (white-box) e con l’analisi di copertura del codice. Innanzitutto abbiamo verificato come fosse la Coverage raggiunta con solo i test black-box, ottenendo la seguente situaziione :





E’ evidente che determinate parti non sono state testate in modo adeguato. Ci siamo inizialmente focalizzati sulla stesura dei test sui due if a riga 27 e 31. Abbiamo usato l’approccio proposto dal MC/DC coverage criterion, possibile poiché si parla di output booleani.

A = RequiresNumber

B = password.matches

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | A | !B | A && !B |
| 1 | T | T | T |
| 2 | T | F | F |
| 3 | F | T | F |
| 4 | F | F | F |

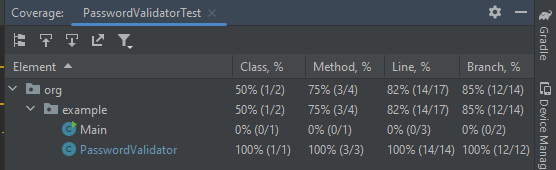
Abbiamo le seguenti coppie :

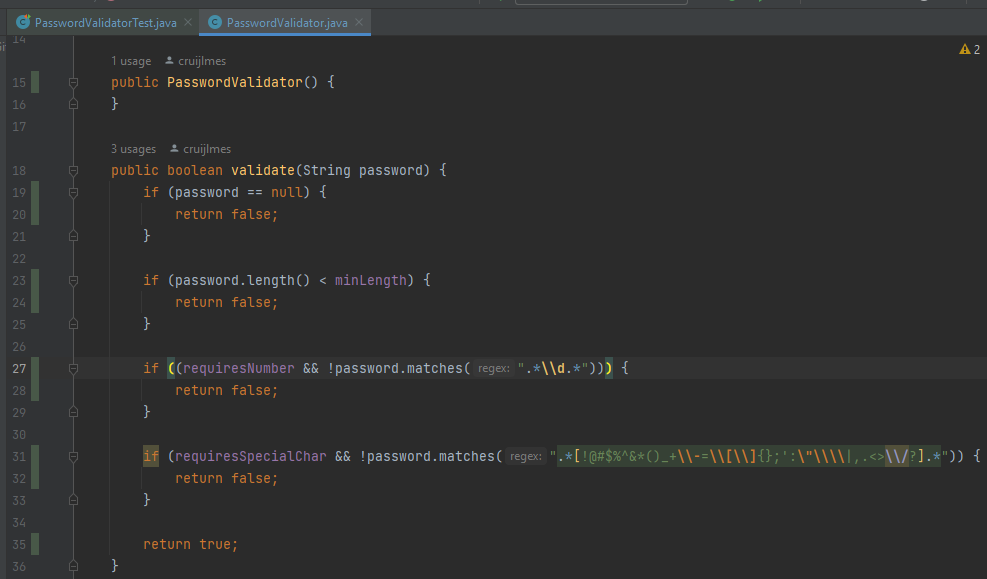
A : {1, 3 }

B : {1, 2}

Sono stati dunque presi in considerazione T1, T2, T3.

Con questi 6 test nuovi, 3 per ognuno dei due if presi in esame, abbiamo ottenuto i seguenti risultati:





Siamo quindi soddisfatti del risultato, ma non essendoci un limite ai test white box, abbiamo deciso di scriverne altri per un senso di completezza. E’ qui che abbiamo riscontrato quello che secondo noi è un errore di progettazione della classe presa in esame.

Se consideriamo la T4 della tabella precedente notiamo che seppur requiresNumber (o carattere speciale) sia possibile impostarlo come false, e quindi non richiesto, se inserito il numero (o carattere speciale) nella password si avrà comunque una password definita valida. Il nostro dubbio allora è il seguente: deve esserci coerenza tra il valore del booleano e la presenza o meno del carattere che rappresenta? Una password per essere sicura dovrebbe contenere almeno 1 numero ed 1 carattere speciale, e quindi riteniamo che se il booleano sia settato su false allora non dovrebbe esserci il numero/carattere speciale. Non riteniamo corretto verificare che un numero o un carattere speciale sia richiesto, e nel momento in cui non sia richiesto ignorare la sua presenza o meno. Abbiamo quindi deciso di cambiare i due if come segue :

prima->

if ((requiresNumber && !password.matches(".\*\\d.\*"))) {  
 return false;  
}

dopo->

if ((requiresNumber && !password.matches(".\*\\d.\*")) | (!requiresNumber && password.matches(".\*\\d.\*"))) {  
 return false;  
}

prima->

if (requiresSpecialChar && !password.matches(".\*[!@#$%^&\*()\_+\\-=\\[\\]{};':\"\\\\|,.<>\\/?].\*")) {  
 return false;  
}

dopo ->

if ((requiresSpecialChar && !password.matches(".\*[!@#$%^&\*()\_+\\-=\\[\\]{};':\"\\\\|,.<>\\/?].\*"))   
| (!requiresSpecialChar && password.matches(".\*[!@#$%^&\*()\_+\\-=\\[\\]{};':\"\\\\|,.<>\\/?].\*"))) {  
 return false;  
}

Come è possibile notare dalla prossima tabella, l’unico cambiamento è stato apportato a T4.

A = RequiresNumber

B = password.matches

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test case | A | !B | !A | B | (A & !B) |  (!A & B) |
| 1 | T | T | F | F | T |
| 2 | T | F | F | T | F |
| 3 | F | T | T | F | F |
| 4 | F | F | T | T | T |

Abbiamo introdotto un test T4 per entrambi gli if per verificare che fosse tutto okay, e funziona tutto perfettamente.

Task2

L’obiettivo del seguente task è stato quello di raggiungere il 100% code coverage con il minor numero di test possibili.

Innanzitutto abbiamo dedicato del tempo alla corretta comprensione del metodo sui cui stiamo lavorando.

L'obiettivo del programma è contare il numero di occorrenze di una determinata parola all'interno di una stringa.

Gli input sono 2 stringhe: “str” e “word”.

L'output è un intero che rappresenta il numero di occorrenze di “word” in “str”.

Il programma prende una stringa in input, sostituisce i caratteri numerici con 'X', quindi divide la stringa in sottostringhe separate da spazi. Conta quante volte la parola “word” appare tra queste sottostringhe e restituisce il conteggio.

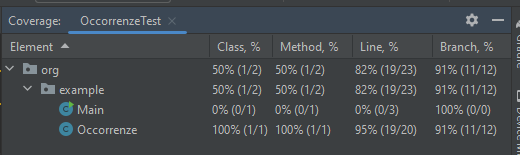
Successivamente abbiamo provato il programma con diversi inputs per verificare il comportamento di quest’ultimo al variare di questi. Abbiamo iniziato testando gli input individualmente e successivamente abbiamo identificato i test che formeranno la suite. Testando gli input come nulli e vuoti, abbiamo notato che “word” non è in alcun modo controllata circa cosa contiene; quindi da un punto di vista del code coverage, con minor test, non ci sono test su “word” almeno per il white-box. Abbiamo quindi, inizialmente, identificato 4 casi di test su cui concentrarci:

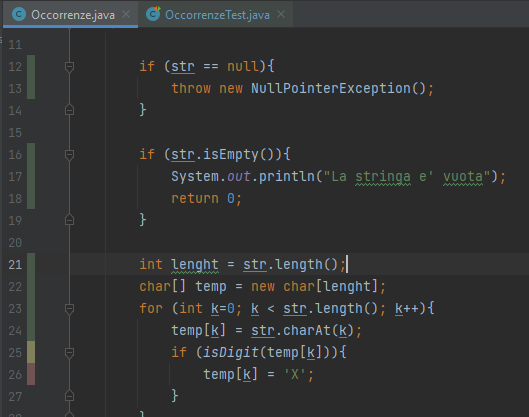
1. Input str non vuota, con almeno una occorrenza di word.

2. Input str non vuota, senza occorrenze di word.

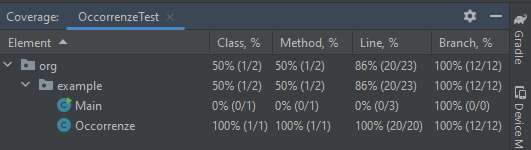
3. Input str vuota.

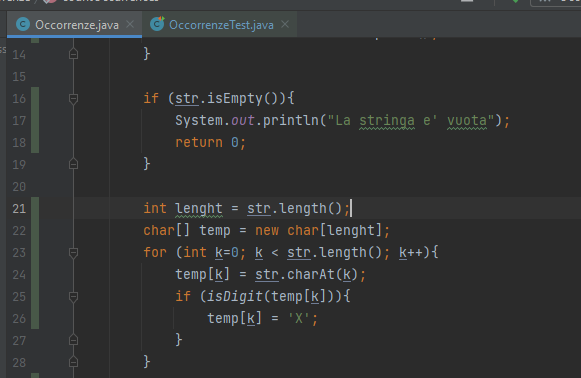
4. Input str nullo.





Avendo scritto questi test e testato il code coverage, abbiamo notato che non comprendeva le righe riguardanti la sostituzione del numero in “str”. Abbiamo quindi mantenuto sempre questi 4 test ma abbiamo aggiunto nella “str” un numero. Con questo accorgimento siamo riusciti in solo 4 test ad ottenere un 100 % code coverage.





Ci teniamo a specificare che nonostante questa percentuale, i test non sono assolutamente esaustivi. Non sono infatti considerati diversi casi limite e l’assenza di controllo su “word” comporta un errore progettuale notevole.

Innanzitutto in presenza di “word” come null il programma crusha. Compilato “word” come spazi, il programma ritorna 0 match. Se “word” è inizializzata come vuota il programma non trova match, almeno affinché “str” non inizi con uno spazio; in quel caso conta gli spazi iniziali come parole e quindi trova 1 match per ogni spazio.

Ci teniamo che a specificare che nonostante “word” sia letteralmente definibile come parola, il suo campo può essere compilato come caratteri speciali, numeri e punteggiatura. In questi 3 casi il programma funziona normalmente seppur troviamo insensato compilarlo come numero nel momento in cui il programma sostituisce tutti i numeri dalla frase in cui cercare il match. Con dei controlli su “word” sarebbe possibile evitare tutte queste ambiguità e problematiche.

Altri due casi limiti sono la compilazione di “str” come solo numeri e come solo spazi. In entrambi i casi il programma si comporta in modo adeguato nella misura che non crusha e non trova match insensati (partendo dal presupposto che “word” sia diverso da null). E’ interessante notare come se assegnato null a “word”, il programma non crusha se “str” è inizializzata con spazi.

Altro limite del programma è la sua impossibilità nel distinguere una parola seguita da punteggiatura. Nell’ esempio : “Ho fame.” -> str “fame” -> word

Non ci saranno match, perché il programma considera “fame.” come un’unica parola.

Il programma non è case-sensitive.