IDEAL Test Plan

Corso di Ingegneria Gestione ed Evoluzione del Software | IDEAL

Domenico Antonio Gioia
d.gioia7@studenti.unisa.it
Matricola: 0522501518

Giovanni Scorziello
g.scorziello5@studenti.unisa.it
Matricola: 0522501701

Antonio Scognamiglio
a.scognamiglio32@studenti.unisa.it
Matricola: 0522501496

Link Repository GitHub

1 INTRODUZIONE

Questo documento contiene le informazioni relative alla pianificazione delle attività di testing eseguite per il progetto. Sono indicate, inoltre, le funzionalità che non sono state sottoposte a test con le motivazioni per ogni scelta.
Sono anche spiegate le varie strategie di testing applicate, insieme ad una descrizione degli strumenti di supporto utilizzati.
L'approccio di testing consiste nell'assumere di testare tutte le funzionalità, salvo poi definire, eventualmente, un insieme di funzionalità che non verrà sottoposto a test specificando il motivo della scelta, come illustrato nel Capitolo 4.

2 PANORAMICA DEL SISTEMA

Il sistema è progettato per assistere gli sviluppatori nello sviluppo di programmi Java e C#. Può analizzare i file di codice sorgente per individuare i linguistic antipattern definiti da Arnaoudova, Di Penta, and Antoniol. L'utente può analizzare una singola cartella o un file e specificare se i file sono di test o no. Vengono controllati elementi come classi, metodi e variabili e le violazioni alle regole vengono registrate in un file CSV. Esistono, inoltre, alcune regole specifiche che vengono controllate per i casi di test.
Le change request proposte, includono di permettere al software di analizzare file di codice sorgente Python, oltre che semplificare le fasi di installazione, configurazione, ed esecuzione del tool.
Tuttavia, è necessario sottolineare, nell'ambito di questo specifico documento, che il progetto originariamente non prevede alcuna attività di testing, e che quindi tutti i test sono stati implementati per rendere più robuste le attività di manutenzione ed evoluzione.

3 DOCUMENTI DI RIFERIMENTO

Nell'elaborazione dei casi di test si è fatto riferimento ai seguenti documenti:

- **Reverse Engineering Document**: le attività di testing sono elaborate in relazione ai requisiti funzionali e non funzionali pre-esistenti del software, oltre che rispetto ai modelli e ai diagrammi contenuti in questo documento.
- **Package Diagram**: la progettazione e la suddivisione delle suite di test avviene rispetto alla decomposizione del sistema in package.

4 FUNZIONALITÀ NON SOTTOPOSTE A TEST

Non tutti i metodi presenti nel progetto verranno sottoposti a test. Di seguito, sono indicati i metodi che non saranno soggetti a test:

- **src.service.config_reader.ConfigReader:**
  - *__get_config_setting()*: Il metodo non è testato in quanto non è utilizzato nel contesto del progetto. È da rilevare che esiste già il metodo *get_config_setting* all'interno della classe *Util* nella directory *common*, il quale viene impiegato attivamente nel progetto per svolgere la medesima funzione. Inoltre, è importante sottolineare che il metodo in questione presenta un'anomalia nella sua firma, poiché non presenta il parametro *self*.
  - *read_config():* Il metodo non è testato in quanto non è deducibile il suo reale utilizzo.
- **src.nlp.pos_tag.get_tag_text():** il metodo equivale all' esecuzione di uno switch statement, motivo per cui non viene sottoposto a testing di unità in quanto banale.
- **src.common.enum.py:** Il modulo è composto da diverse enumerazioni, utilizzate all'interno del progetto, ovvero:
  - *LanguageType*: Per la distinzione del linguaggio con il quale è scritto il progetto
  - *FileType*: Per la distinzione del tipo di file all'interno del progetto (se si tratta di un file di testing o di progetto)
  - *IdentifierType*: Per la distinzione del tipo di costrutto che si sta analizzando, se si tratta di un metodo, un attributo, ecc..

  Risulta quindi inutile la creazione di casi di test per questo modulo, in quanto non vi è alcuna logica complessa.
- **src.common.logger.setup_logger:** Il metodo non verrà modificato durante la fase di manutenzione ed evoluzione del progetto, non viene quindi considerato durante la fase di testing
- **src.common.util_parsing.py:**
  - *get_all_exception_throws(), get_all_return_statements(), get_all_function_calls(), get_all_conditional_statements()*: Questi metodi non hanno alcuna documentazione e dal codice non è facilmente deducibile il loro comportamento atteso, ne il loro funzionamento, che, tuttavia, è legato a srcML, il quale, inoltre, risulta essere non funzionante dai relativi test d'integrazione svolti.
- **src.model.***: questo package contiene classi che rappresentano entità del problema nel contesto di IDEAL, e i file non verranno sottoposti a test in quanto consistono principalmente di metodi *getter* e *setter*.
- **src.model.entity.construct_hierarchy()**: il metodo consiste di circa 270 righe di codice. Per questo motivo, è complesso testarlo in unità; in integrazione risulterebbe complicato individuare la causa di eventuali problemi, oltre che stabilire le categorie di input da testare. Oltretutto, è molto complesso anche scomporla in sottofunzioni oppure tentare di documentarla.

5 PASS/FAIL CRITERIA

Ogni caso di test è composto da dati di input e oracolo, ovvero l'esito atteso. Un caso di test verrà considerato come avente successo se dato un input il risultato è uguale al risultato atteso (pass), mentre come fallito (fail) altrimenti. Le attività di testing saranno considerate valide al raggiungimento di una percentuale minima di branch coverage dell' 80%, raggiunta considerando solo le funzionalità da testare, escluse dal Capitolo 4.

6 APPROCCIO

6.1 Test di unità

Per il testing di unità la strategia utilizzata consiste nel testare i metodi di alcuni moduli che compongono il sistema, in maniera isolata. I casi di test saranno definiti attraverso un approccio white-box che permette di tenere conto dei dettagli implementativi per la progettazione dei vari test. I riferimenti a questo e altri documenti, oltre che specifica documentazione per i casi di test sarà inclusa direttamente nel codice tramite l'utilizzo di commenti in formato *docstring*.
Gli obbiettivi di questo tipo di testing riguardano sia il testing del comportamento del software rispetto agli input dell'utente, sia quello rispetto a come è stato inizialmente implementato il sistema, cercando, quindi, errori nel codice che possono essere stati inseriti durante la sua prima implementazione.

6.2 Test di integrazione

I test di integrazione sono stati progettati ed implementati subito dopo le attività di testing di unità. Questo permette l'esecuzione ordinata e divisa delle varie attività di testing per seguire un approccio *bottom-up* nel testing d'integrazione: questo perchè l'esercitazione del codice, dapprima in isolamento, e successivamente integrando i vari componenti man mano, consente di testare il comportamento dei vari moduli del sistema ed eventualmente identificare più facilmente quali componenti includono dei problemi.

6.3 Test di sistema

Idealmente le attività di testing di sistema dovrebbero essere automatizzate, ma siccome non è stato identificato un tool che potesse essere adatto a questo progetto, verranno eseguite manualmente. Nello specifico si progetteranno dei file di input, verrà eseguito il programma e i risultati verranno ispezionati manualmente per assicurare il corretto funzionamento del software.

Dal momento che effettuare il testing manualmente richiede un effort elevato, è stato deciso, per non togliere tempo all'implementazione delle modifiche, di eseguirlo solo dopo detta implementazione. Anche in questo caso, come per gli altri test, sono disponibili i dettagli dei casi di test pianificati negli allegati presenti nel Capitolo 10.

6.4 Strumenti di supporto

Per automatizzare l'esecuzione dei casi di test è stata usata la libreria ***pytest***. Per l'implementazione dei casi di test sono state seguite le pratiche raccomandate dal tool, ovvero, l'utilizzo di fixture per la preparazione dell'ambiente di test, l'uso delle asserzioni per la verifica dei risultati e la partizione delle categorie per garantire una copertura più accurata e completa. Questo permette di assicurare un alto livello di qualità e affidabilità del software.
Altri strumenti utilizzati per garantire la qualità del codice sono, ad esempio, *autopep8* per imporre il code style PEP8[1], e GitHub Actions di GitHub per realizzare una pipeline di Continuous Testing che permette di trovare eventuali test falliti tempestivamente e caricare le informazioni di coverage su CodeCov.
Infine, durante l'implementazione di tutti i casi di test, verrà usata la funzione di code review di GitHub per l'integrazione del nuovo codice nella repository, in modo da ispezionare anche manualmente il codice sorgente.

7 TEST CASE SPECIFICATION

I test case in forma tabellare possono essere visualizzati tramite l'allegato 1 nel Capitolo 10, mentre le specifiche degli stessi si trovano nell'allegato 2.

8 TESTING SCHEDULE

Le attività di testing illustrate in questo documento sono state pianificate in modo da dover essere eseguite prima, durante e dopo lo sviluppo delle modifiche per l'implementazione delle change request. Nello specifico, i casi di test riguardanti il codice originale verranno progettati, documentati in questo documento, scritti ed eseguiti prima dell'implementazione delle modifiche, dal momento che il progetto non include nessun test, e successivamente in maniera simultanea all'inserimento del nuovo codice nel progetto, permettendo la conduzione di test di regressione. Una volta concluse le modifiche tutti i test saranno eseguiti per garantire il corretto funzionamento e produrre i report finali, visualizzabili nel Capitolo 10.

9 RISULTATI

Come mostrato nell'allegato 4 del Capitolo 10, un unico test continua a non passare, ed è *test_split_word_tokens_mixed()*, con id *TC-NLP-6.3*. Nonostante i diversi tentativi, la mancanza di documentazione nel codice originale non permette di determinare esattamente quale sarebbe il comportamento atteso della funzione; il nome sembra suggerire che essa dovrebbe estrarre e tokenizzare le parole da una stringa, per cui se quest'ultima contiene dei simboli non alfabetici dovrebbero essere rimossi dal risultato, e questo accade con alcuni simboli, ma non tutti. Bisogna anche considerare il contesto in cui viene utilizzata questa funzione che potrebbe rendere questo problema un falso positivo nel caso in cui caratteri non alfabetici non vengano mai passati come argomento della funzione. In attesa di poter risolvere definitivamente il problema, è stata anche aperta una *issue* nella repo *SCANL/ProjectSunshine* per richiedere chiarimenti in merito al comportamento della funzione, come mostrato in Figura 1.

Inoltre è stato riscontrato che due ulteriori test di unità, denominati *test_get_tag_text_verb_lowercase()* e *test_get_tag_text_verb_mixed_case()*, riportavano un esito di fallimento; successivamente, durante una revisione approfondita del codice sorgente, è emerso che mancava un meccanismo di normalizzazione della stringa all'interno del metodo. Dopo aver individuato

[1] https://peps.python.org/pep-0008/

Figure 1: Issue aperta nella repo SCANL/ProjectSunshine

l'errore, è stato eseguito un intervento correttivo per risolvere il problema.

10 ALLEGATI

(1) Tabelle di partizione dei test case
(2) Tabelle di specifica dei test case - Test case specification
(3) Test Report Pre-Implementazione
(4) Test Report Post-Implementazione
(5) Test Report di Sistema

REFERENCES

Arnaoudova, Venera, Massimiliano Di Penta, and Giuliano Antoniol. "Linguistic antipatterns: What they are and how developers perceive them". In: *Empirical Software Engineering* 21 (2016), pp. 104–158.