Test Documentation

cASpeR su Maven

Riferimento	
Data	19/06/2022
Destinatario	Prof. De Lucia, Dott. Di Nucci, Dott. Manuel De Stefano, Dott. Emanuele Iannone
Presentato da	Lerose Ludovico, Milione Vincent
Approvato da	

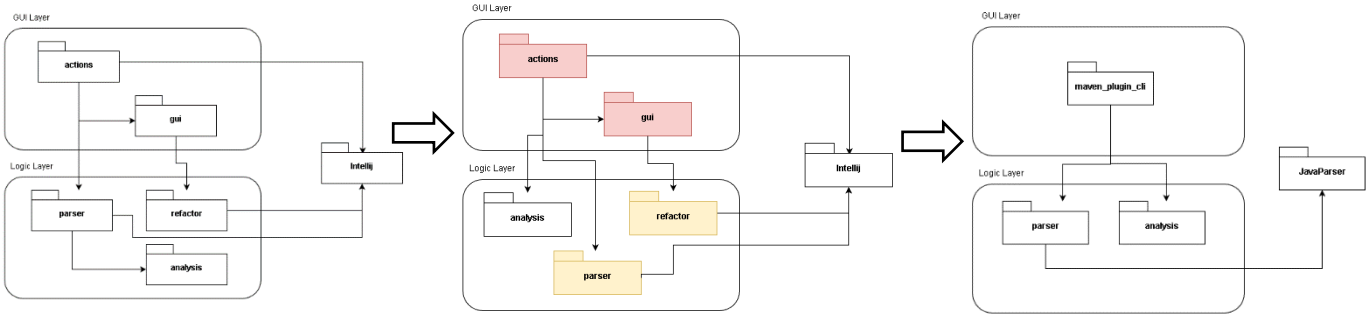
Sommario

1. Introduzione e Planning 3
1.1 Testing Strategies MR1 3
1.2 Testing Strategies MR2 4
1.3 Testing strategy 5
Testing parser 5
Testing di sistema 5
1.4 Testing Technologies and Procedures 5
Testing parser 5
Testing di sistema 6
2. Test Frames 6
2.1 Test d'unità del parser 6

1. Introduzione e Planning

Lo scopo del documento è quello di illustrare la pianificazione delle attività di testing e le strategie di testing riguardanti l'applicazione cASpER reingegnerizzata come maven plugin.

Nelle sezioni di analisi e progettazione del documento di progetto sono state proposte e analizzate le varie soluzioni alle introduzioni delle modifiche. Infine, sono state decise le azioni da intraprendere e la vista del sistema finale, a seguito dell'introduzione di MR1 e MR2, è la seguente.



In particolare, notiamo che a seguito delle modifiche da eseguire:

- l parser verrà modificato ed isolato mediante un'operazione di extract class refactoring
- Il modulo "refactoring" non verrà trasferito, in quanto sarà implementato in un successivo incremento
- La parte di analisi non richiede di essere testata in quanto non viene toccata

L'obiettivo di queste attività di testing è assicurarsi che il plugin versione maven si comporti in maniera simile alla versione originale. In particolare, vogliamo che la parte responsabile dell'analisi statica comunichi gli stessi risultati della versione originale. A seguito delle informazioni recepite dal documento di progetto e dell'obiettivo illustrato possiamo dedurre che i due moduli che necessitano assolutamente di essere testati sono il parser e il nuovo modulo "maven_plugin_cli". Inoltre, è assolutamente necessario un test di sistema.

1.1 MR1

Data la restrizione dello scope di refactoring dovuta a MR2, le parti effettivamente re-fattorizzate sono nel modulo di analisi. Ricordiamo che MR1 si prepone di eliminare moduli "polluted". Per moduli "polluted" s'intende moduli con delle funzionalità esistenti nel codice, ma mai usate.

Come discusso nella sezione 3.1 e 2.1 del documento di progetto, le parti da re-fattorizzare sono code smell e le loro relative strategie. Grazie all'applicazione dello strategy pattern, rimuovere un code smell con le sue relative strategie o delle strategie in particolare è piuttosto semplice. In più le modifiche non impattano gli altri code smell, i loro algoritmi di detection e il loro uso nella fase di analisi statica. Bensi chi usa i code smell e le loro relative strategie sono impattati.

Come accennato nelle parti finali della sezione 3.1 del documento di progetto, i cambiamenti si propagano in linea teorica verso il parser e l'interfaccia grafica. Questo non introduce ulteriori oneri alle attività di testing di MR1 perché:

- Durante le attività di reverse engineering, si è notato che il parser non avvia né l'analisi statica per l'individuazione dei nuovi code smell implementati né applica i nuovi algoritmi di history implementati. Il codice responsabile è stato commentato. Non dovendo modificarlo, non è necessario testarlo in virtù di MR1.
- Come indicato nella sezione 4.2 della progettazione delle modifiche le informazioni presentate saranno solo un subset delle informazioni presentate nel tool originale. In seguito all'intervento di MR2, sarà necessario confrontare queste informazioni prodotte dalla versione maven con quelle del tool originale attraverso un test di sistema.

Quindi le modifiche rimangono locali al modulo di analisi, ma non impattano in nessun modo le componenti esistenti all'interno. Pur essendo dotati di una test suite per il modulo di analisi, non è comunque necessario un test di regressione, ma lo si può fare per completezza.

1.2 MR2

Come si è visto dalle figure iniziali, l'intervento di reengineering coinvolge più layer del sistema.

I due approcci al testing a cui si è pensato per MR2 sono gli approcci bottom-up e sandwich. L'approccio top-down potrebbe avere senso siccome le modifiche sostanziali sono alla parte interattiva del tool. Si deve però considerare che l'uso della nuova libreria – Java parser – è in grado di stravolgere il funzionamento della classe. Siccome tutta l'attività di analisi statica inizia a seguito del parsing del progetto software, l'approccio top down non può essere adottato. È necessario testare il parser in isolamento per verificare la presenza di malfunzionamenti.

Dunque, non restano altro che i due approcci inizialmente indicati. L'approccio sandwich sarebbe l'ideale per i nostri scopi. Sia la logica del parser che la logica di presentazione potrebbero essere testate in isolamento. Dato che il modulo di analisi non ha bisogno di essere ritestato in isolamento, possiamo poi procedere direttamente con l'integrazione delle tre componenti ed eseguire un test di sistema complessivo.

Nell'approccio bottom-up, l'idea sarebbe quella di testare in isolamento la logica del parser e poi procedere con l'integrazione con la logica di presentazione. Come già accennato, il parser è responsabile per strutturare l'input per il modulo di analisi. Di conseguenza il modulo maven_plugin_cli non restituirebbe nulla in output se non viene integrato anche con la parte di analisi. Una volta testato in isolamento il modulo parser, si effettuerebbe immediatamente un test di sistema.

Bottom-Up		**Sandwich**	
Pro	Contro	Pro	Contro
Riduciamo i tempi di testing	Non viene testata la logica di presentazione in isolamento	Tutti i moduli introdotti e modificati vengono testati in isolamento	Maggiori tempi richiesti per il testing
	+ Rischi di malfunzionamento nella logica di presentazione	- Rischi di malfunzionamento nella logica di presentazione	

Bisogna tenere conto che l'effort per realizzare dei casi di test per il parser e per il test di sistema èalto per via della scarsa possibilità di automatismo. Questi casi di test non esistono nella versione originale di cASpER. Di conseguenza, non solo bisognerebbe ideare i casi di test per la versione maven, ma bisognerebbe anche avviarli sul sistema originale per estrarre i nostri oracoli.

Dovendo limitare l'effort da impiegare considerando che i membri nel gruppo sono solo due, si è preferito l'approccio bottom-up. L'approccio sandwich avrebbe senso nel caso in cui siano richiesti più livelli di

integrazione, in modo da non ritardare troppo il testing dei moduli di “coordinamento”. Questo non è il caso nostro. Subito dopo il testing del parser, possiamo testare almeno una volta la logica di presentazione a livello di sistema.

1.3 Testing strategy

Di seguito riportiamo le strategie di testing ai due livelli identificati. Quello unitario per il parser e quello a livello di sistema.

Testing parser

Come accennato verso la fine del paragrafo introduttivo, non abbiamo a disposizione nel tool originale un test per il parser. A seguito della sezione 4 del documento di progetto, si può concludere che del modulo parser è necessario verificare la presenza di errori nel metodo parse() della classe ProjectParser.java.

Per affrontare il testing funzionale del parser si è deciso di applicare la tecnica black-box (quindi un approccio sistematico) del **category partition**, in quanto sembra essere la più flessibile tra tutte le strategie black-box che si sono studiate. Flessibilità sia in termini della applicazione sul particolare dominio di input – i progetti software maven – sia in termini del numero dei casi di test da produrre.

L’applicazione della tecnica è vista nella sezione 2.1 di questo documento.

Uno dei problemi maggiori nel testare il parser è che la versione originale implementa due responsabilità, il parsing del progetto e l’avvio dell’analisi statica. Ciò rende l’estrazione automatica degli oracoli un po’ più problematica. Nella versione Maven, siamo interessati solo a verificare il corretto funzionamento della prima responsabilità rispetto a come lo fa il tool originale. Le soluzioni trovate sono due:

1) Estrazione manuale degli oracoli dal tool originale
2) Realizzazione di casi di test che saltano l’attività di analisi

In caso non sia possibile trovare un approccio automatico, daremo il via all’estrazione manuale degli oracoli.

Testing di sistema

Seppure non esistano dei veri e propri test funzionali a livello di sistema, esistono però i test a livello di integrazione del modulo di analisi nel tool originale. Questi possono essere facilmente scalati a livello di sistema e usati con lo scopo di verificare il comportamento del tool maven in confronto a quello originale.

Per tanto si è deciso di prendere la test suite esistente ed adattarla a livello di sistema.

1.4 Testing Technologies and Procedures

Di seguito riportiamo le tecnologie e procedure da seguire all’esecuzione dei test case ai diversi livelli. Gli step indicano come generare il test report finale.

Testing parser

La realizzazione dei casi di test avverrà attraverso l’uso della tecnologia JUnit 4. In entrambi le versioni del plugin verranno introdotti i casi di test con questa tecnologia. Questo permetterà di automatizzare le attività di testing execution e la generazione del test report.

Per la generazione dei test report della versione originale seguiamo gli stessi identici passaggi indicati nella sezione introduttiva del Test Documentation del plugin originale.

È previsto che i casi di test vengano eseguiti sull'IDE Intellij 2019.3 Ultimate Edition. Dato che il progetto è in Maven, l'esecuzione del test avverrà attraverso la fase del ciclo di vita chiamato "test". I risultati dei test saranno poi raccolti in un test report generato automaticamente dall'IDE in formato HTML. Identifichiamo e dettagliamo i seguenti step che devono essere presi per la fase di esecuzione e generazione del test report:

1) Aprire il progetto cASpER versione plugin maven con IDE Intellij 2019.3
    a. Se non è stato ancora importato il progetto, allora s'importa
2) Aprire la finestra laterale di Maven su Intellij
3) Aprire la tendina "Lifecycle"
4) Doppio click su opzione "test"
5) Al termine dell'esecuzione dei test, cliccare sull'icona dell'export dei risultati
6) Scegliere formato HTML e salvare con nome "Test Report – cASpER_plugin_maven.html"

Ricordiamo che consideriamo un'anomalia da documentare solo i casi in cui un caso di test del parser:

- Passa nella versione maven, ma non nella versione Intellij
- Passa nella versione Intellij, ma non nella versione maven

Testing di sistema

Non avendo una conoscenza dei tool per effettuare test di sistema per plugins, il test sistema sarà condotto in modo più manuale.

Per la versione Intellij, il test di sistema sarà condotto usando il task gradle "runIDE". Tale task consente di costruire un'istanza del programma Intellij 2019.3 con il plugin cASpER. Da qui proveremo ad eseguire il plugin sui progetti software che si deciderà di usare come input di un test case. Di conseguenza, i risultati del test case dovranno essere riportati manualmente in un documento apposta.

Per la versione Maven, il test di sistema verrà leggermente automatizzato mediante l'uso di un potenziale bashscript.

2. Test Frames

Introduciamo in questa sezione il modo in cui è stato applicato category partition. In ogni sottosezione vedremo le categorie identificate per i parametri e loro relative scelte e vincoli. In seguito, saranno riportati i test frame conformi ai vincoli che si sono decisi di applicare. Non verrano specificati in questo documento i casi di test. Essi saranno documentati insieme ai casi di test JUnit che verrano creati.

2.1 Test frame parser

Per il test unitario del parser si è pensato di utilizzare la tecnica di testing funzionale category partition.
Il parametro è unico ed è il progetto software Java da essere "parsato".

Precondizione: Progetto software Java Maven.

Postcondizione: Lista dei package bean istanziati con il nome del rispettivo package e i contenuti testuali delle classi contenute in esso.

Numero di packages
- 0 [property nopackages]
- 1 [property singlepackage]
- molti [property multipackage]

Numero di classi
- 0 [if nopackages]
- 1 [if singlepackage] [property singleclass]
- molti [if singlepackage] [if multipackage] [property manyclasses]

Buildabile
- si
- Errore lessicale presente del progetto [if singlepackage] if[singleclass]
- Errore sintattico presente in una classe del progetto [if singlepackage] [if singleclass]
- Errore semantico presente in una classe del progetto [if singlepackage] [if singleclass]

Numero di gerarchie
- 0
- 1 [if manyclasses]
- molti [if manyclasses]

Combinazioni

P_TC1: NumPackages[0], **NumClasses**[0], **Buildable**[si], **NumHierarchy**[0]

P_TC2: NumPackages[1], **NumClasses**[1], **Buildable**[si], **NumHierarchy**[0]

P_TC3: NumPackages[1], **NumClasses**[1], **Buildable**[errore lessicale], **NumHierarchy**[0]

P_TC4: NumPackages[1], **NumClasses**[1], **Buildable**[errore sintattico], **NumHierarchy**[0]

P_TC5: NumPackages[1], **NumClasses**[1], **Buildable**[errore semantico], **NumHierarchy**[0]

P_TC6: NumPackages[1], **NumClasses**[molti], **Buildable**[si], **NumHierarchy**[0]

P_TC7: NumPackages[1], **NumClasses**[molti], **Buildable**[si], **NumHierarchy**[1]

P_TC8: NumPackages[1], **NumClasses**[molti], **Buildable**[si], **NumHierarchy**[molti]

P_TC9: NumPackages[molti], **NumClasses**[molti], **Buildable**[si], **NumHierarchy**[0]

P_TC10: NumPackages[molti], **NumClasses**[molti], **Buildable**[si], **NumHierarchy**[1]

P_TC11: NumPackages[molti], **NumClasses**[molti], **Buildable**[si], **NumHierarchy**[molti]