UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

Strumenti e tecniche per l'automazione del testing black box basato su classi di equivalenza

Anno Accademico 2017/2018

Candidato:

Davide Feliciello

matr. N46000661

[Dedica]

Indice

Introduzione

Nelle ultime decadi la tecnologia ha avuto un'evoluzione sorprendente. Sono stati ideati migliaia di smartphone, personal computer, smartwatch e negli ultimi anni ha preso piede l'Internet Of Things. I software ci permettono di interagire con tali dispositivi e rivestono un ruolo sempre più critico, per cui è necessario che essi funzionino in modo corretto e affidabile. Un software si può dire funzioni correttamente se rispetta i requisiti funzionali, in caso contrario potrebbe essere soggetto a bug. Quest'ultimo deriva da un errore di progettazione nel software e può essere sintomo di vulnerabilità poiché gli hacker potrebbero ottenere informazioni sensibili e sfruttarle in modo malevolo. In passato è stato anche causa di numerose catastrofi come l'esplosione del razzo Ariane 5 per via di un'eccezione non gestita. Episodi come questo mettono in evidenza l'importanza di avere software sicuro, affidabile e corretto.

Il failure è un comportamento di un programma discordante dai requisiti e si verifica quando non fa quello che l'utente si aspetta mentre un bug è una sequenza di istruzioni, che, quando eseguita con una particolare combinazione di dati di ingresso, genera un malfunzionamento.

Il testing è una procedura usata per verificare il comportamento a run-time di un programma e consiste nell'esecuzione del codice associato a degli input specifici, si può dire che esso ha successo nel caso in cui si trova uno o più errori nel codice testato.

Il testing rappresenta la parte di analisi dinamica del processo di verifica del software, la parte complementare, ossia quella statica, è invece rappresentata dall'ispezione.
Il principale ostacolo al testing è sintetizzato nella Tesi di Dijkstra: il testing può indicare la presenza di errori, ma non ne può garantire l'assenza. Purtroppo nessun testing può ridurre a zero la probabilità di non avere bug o failure, in quanto le possibili combinazioni di valori di input validi sono enormi, e non possono essere riprodotte in un tempo ragionevole. Quindi il testing esaustivo è praticamente impossibile. Tuttavia un buon testing può rendere la probabilità di malfunzionamenti abbastanza bassa da essere accettabile dall'utente, l'obiettivo primario deve essere ottenere il miglior rapporto possibile fra la qualità del prodotto e il tempo e le risorse impiegate nelle attività di controllo. L'attività di testing si reputa conclusa in base a criteri di costo, tempo o statistici. Per il testing, in genere, si investe tra il 30% e il 50% del budget totale del progetto e una gran quantità di tempo allo scopo di evitare futuri refactoring del sistema e, di conseguenza, far risparmiare soldi e ulteriore tempo impiegato. Un testing fatto bene permette di ridurre in modo sostanziale i costi di manutenzione del programma sviluppato.

La prima parte di questa tesi sarà dedicata alla descrizione delle differenti modalità di testing e in particolare fra queste ci focalizzeremo su quella black-box basata su classi di equivalenza. In seguito verrà presentato uno strumento molto utile per generare dati di test chiamato EcFeed che si integra molto bene con JUnit, un framework di testing di unità per java, che permette di automatizzare la procedurà di test. Successivamente verrà presentato un tool, anch'esso ben integrato con JUnit, per valutare l'efficacia dei test svolti in modalità black-box: EclEmma. In questo modo possiamo progettare i test in maniera black-box e valutare l'esito in white-box per avere una misura oggettiva dell'efficacia assoluta e relativa dei test progettati. Per concludere verrà mostrato un esempio in cui metteremo alla prova quanto appreso.

Capitolo 1: Modalità di Testing

Si definisce caso di test una tripla <input,output,ambiente> dove l'ultimo parametro descrive l'ambiente in cui verrà eseguito il test. Una test suite è una sequenza di casi di test, all'aumentare del numero di test effettuati aumenta la probabilità di rilevare malfunzionamenti ma bisogna tener conto che essi hanno un costo.

Le tecniche di testing possono essere divise in due classi principali: criteri funzionali e strutturali. I primi richiedono l'analisi degli output generati dal sistema in risposta ad input definiti sulla solo conoscenza dei requisiti del sistema mentre quelli strutturali sono fondati sulla conoscenza della struttura del software.

Un aspetto importante nella progettazione dei test è la necessità di definire un oracolo, esso genera i risultati attesi con cui verranno confrontati i risultati ottenuti dai casi test allo scopo mettere in luce eventuali malfunzionamenti. L'oracolo risulta di difficile definizione in molti casi, specialmente quando i test case sono progettati secondo criteri strutturali e non si hanno indicazioni sui risultati che si devono attendere. E' importante che sia l'oracolo che il software condividano le stesse condizioni di test ovvero input e ambiente di test.

Il testing può essere svolto in tre modalità: white-box, black-box e grey-box. Esse non devono essere viste come tecniche di testing ma come una generica famiglia di tecniche.

- Nel testing white-box il componente software testato risulta trasparente, quindi il tester può visionare il codice e scoprire quale parte di esso potrebbe dare problemi. Il comportamento ad un dato ingresso si analizza osservando come esso "attraversa" il codice. Solitamente questa modalità di testing è svolta dal

programmatore stesso perché richiede conoscenza e comprensione del sistema testato.

- Il testing black-box consiste nell'immettere input e osservare i valori dell'output. Il programma è trattato come una scatola chiusa quindi non si tiene conto del codice sorgente, dello stato e del funzionamento del componente sotto test. I casi di test vengono progettati solo utilizzando l'interfaccia input/output del sistema, la documentazione e l'ambiente di esecuzione. Risulta adatto per codice di grandi dimensioni e a differenza del testing white-box non deve necessariamente essere svolto dallo scrittore del software ma al contrario può essere effettuato da un tester non particolarmente esperto senza conoscenze di linguaggi di programmazione e sistemi operativi. Il principale vantaggio di questo approccio è la possibilità di progettare i casi di test parallelamente allo sviluppo del software poiché essi non dipendono dal codice del programma da testare ma solo dalle specifiche che definiscono in maniera completa le funzionalità del sistema. Di contro questo tipo di testing può lasciare porzioni estese di codice non testate visto che non possiamo misurare la copertura, inoltre il tester ha una limitata conoscenza dell'applicazione e può non testare sufficientemente il sistema. Se le specifiche sono espresse in linguaggio formale questa tecnica risulta facilmente applicabile in quanto si può ricavare in maniera esatta l'insieme degli ingressi del programma.
- Il testing gray-box deriva dall'idea di fondere gli approcci black-box e white-box. Esso prevede di testare il programma conoscendo i requisiti e avendo una limitata conoscenza della realizzazione, per esempio conoscendo solo l'architettura. Una strategia per realizzarlo è quella di progettare i test usando criteri funzionali e successivamente di usare le misure di copertura dei criteri strutturali per valutare l'efficacia del test conoscendo che percentuale del codice è stato testato. Ciò è esattamente quello che faremo in questa tesi.

Esistono varie metodologie per il testing black-box che hanno l'obiettivo di individuare i casi rilevanti che saranno l'oggetto dei test:

- Testing basato sui requisiti. Si progettano vari test per ogni requisito in virtù del fatto che il principio di verificabilità dei requisiti afferma che essi dovrebbero essere testabili, cioè scritti in modo da poter agevolmente progettare casi di test per dimostrarli.
- Testing basato sui casi d'uso. A partire dal diagramma dei casi d'uso per ogni scenario si progettano uno o più casi di test che li eseguano, si mira quindi alla copertura dei casi d'uso e degli scenari.
- Testing basato sulle classi di equivalenza. Si divide il dominio dei dati di input in un numero finito di sottoinsiemi (classi) nei quali si suppone che, in base ai requisiti, il codice si comporti allo stesso modo . Ognuna delle classi costituisce una partizione e possono venire identificate attraverso le specifiche del programma o altra documentazione. Vengono progettati casi di test in modo da provare il programma su valori rappresentanti di ciascuna partizione.

  Questo criterio può essere agevolmente applicato senza gravare eccessivamente sui costi solo se il sistema ha un numero dei possibili comportamenti molto inferiore alle possibili combinazioni degli input. In questo caso non ha senso parlare di oracoli perché i risultati attesi del test sono noti in virtù di come sono costruite le partizioni. Il principale svantaggio di questa metodica è che non sempre il corretto funzionamento del programma su di un valore preso dalla classe d'equivalenza implica la correttezza su tutti gli elementi appartenenti a quest'ultima. Ciò risulta vero molte volte nella pratica ma dipende dalla realizzazione del programma e non può essere verificato sulla base delle sole specifiche funzionali.

  Una possibile suddivisione è quella tra stati validi e non validi per una condizione sulle variabili d'ingresso.
- Testing con classi di equivalenza e valori di frontiera. Rappresenta una variante della tecnica delle classi di equivalenza in cui come dati di test vengono considerati i valori limite di ogni classe d'equivalenza. Si applica efficacemente a sottoinsiemi di insiemi continui come numeri interi e reali. Questo criterio risulta molto usato in altre branche ingegneristiche tipo in meccanica, dove una parte

testata per un certo carico resiste con sicurezza a carichi minori, ma non è applicabile con la stessa efficacia nell'ambito software poiché non è possibile dedurre il comportamento continuo del programma. La giustificazione per l'uso sta nel fatto che i valori boundaries in genere vengono trattati in modo diverso dal programma quindi risulta utile in congiunzione al criterio delle classi d'equivalenza.

- Testing statistico. I casi di test vengono scelti in base alla distribuzione di probabilità dei dati di input del programma quindi il sistema viene testato sui valori di ingresso che risultano più probabili a regime. Un vantaggio di questa tecnica è che, nota la distribuzione di probabilità, la generazione di test case è facilmente automatizzabile. Di contro non sempre abbiamo accesso alla distribuzione di probabilità a partire dalle specifiche ed essa potrebbe non corrispondere alle condizioni operative tipiche del programma.
- Testing a partire dalle tabelle di decisione. Sono uno strumento per la specifica black-box di componenti in cui a diverse combinazioni di ingressi corrispondono uscite/azioni diverse che non devono dipendere da ingressi precedenti né dall'ordine in cui sono effettuate. Le possibili combinazioni degli ingressi devono essere mutamente esclusive e possono essere scritte come espressioni booleane. Nelle colonne della tabella troveremo le combinazioni degli input, le cosiddette varianti, mentre nelle righe i valori delle variabili di input e le azioni.
- Testing a partire dal grafo causa-effetto. Sono dei grafi costruiti a partire dai requisiti o dalle specifiche del programma che legano un insieme di variabili di decisione in ingresso (cause) e di azioni di output (effetti) in una rete combinatoria. Sono un modo alternativo per rappresentare le tabelle di decisione e forniscono una rappresentazione grafica intuitiva anche se al crescere della complessità possono diventare ingestibili. I casi di test si ricaveranno risalendo il grafo a partire dagli effetti ottenendo le cause ovvero le combinazioni delle variabili di ingresso che li generano. Il criterio risolve direttamente il problema dell'oracolo per il modo in cui i casi di test sono ricavati. A volte risulta conveniente costruirlo già durante la

fase di validazione dei requisiti per evidenziare contraddizioni e parti mancanti di logica.

I limiti di queste tecniche emergono per lo più all'aumentare della complessità del componenti o del sistema sotto test. In ambito di sistemi interattivi bisogna considerare anche l'ordine di inserimento degli ingressi quindi non sarà più possibile coprire tutte le possibili combinazioni di input. Nei sistemi real time invece riveste molta importanza l'istante di inserimento degli input quindi si pone il problema di discretizzare l'istante in un insieme di classi limitate.

1.1 Strategie per il testing con classi di equivalenza

Esistono diverse strategie per il testing con classi di equivalenza:

- Copertura minima delle classi di equivalenza. Ogni classe viene coperta da almeno un caso di test quindi il numero minimo di casi di test, nell'ipotesi che tutti gli input siano indipendenti e sincroni, è pari alla cardinalità massima tra le classi di equivalenza. Il vantaggio principale di tale approccio è che abbiamo la massima efficienza perché riusciamo a coprire tutte le classi col minimo numero di casi di test. Di contro nel caso si riesca trovare un difetto risulta difficile risalire alla causa poiché il fallimento può essere dovuto a uno qualsiasi degli ingressi o peggio ancora dalla loro interazione.
- Copertura delle classi di equivalenza adiacenti. Anche qui ogni classe è coperta almeno da un caso di test ma la differenza sta nel fatto che per ogni caso di test ne esiste almeno uno che differisce per una sola classe di equivalenza. Il numero di test case necessario sarà pari al numero di classi di equivalenza. Il vantaggio principale è che, rispetto alla strategia precedente, si hanno più informazioni per il debugging poiché se fallisce un solo test possiamo individuarne un altro adiacente che non fallisce e con buona probabilità il problema sarà dovuto all'unico valore di ingresso differente tra i due test. Ciò però è un caso ideale poiché può accadere che, nonostante il difetto sia legato ad un unico ingresso, non viene selezionata una

coppia di casi di test che differisce solo per quell'ingresso. Infine notiamo che questa strategia è meno efficiente poiché il numero di casi di test cresce col numero di variabili di input.

- Copertura delle n-ple di classi di equivalenza. Vengono testate tutte l'insieme delle k-ple con k<=numero di ingressi. Il numero di test eseguiti sarà pari al prodotto delle cardinalità dei k input aventi più classi di equivalenza. Il caso con k=1 corrisponde alla copertura minima. Dati empirici del NIST (National Institute of Standards and Technology) suggeriscono che spesso le failure del software sono causato da meno di 6 parametri e testare le 2-ple può dare risultati accettabili poiché consente di individuare il 60-90% dei bug ad un costo contenuto. Quindi per la scelta di k bisogna tenere conto del trade-off tra il veloce incremento del numero di test da effettuare e i ritorni che diminuiscono all'aumentare di k poiché si trovano man mano meno difetti. Il testing combinatorio con k>2 in passato è stato abbastanza limitato, principalmente per la mancanza di buoni algoritmi, ma recentemente l'introduzione di nuovi algoritmi lo ha reso di più pratico uso per l'industria.
- Copertura combinatoria delle classi di equivalenza. Può essere visto come una generalizzazione del caso precedente con k=numero di ingressi quindi prevede la copertura di tutte le combinazioni delle classi di equivalenza. Questa strategia risulta la meno efficiente possibile poiché il numero di test è pari alla produttoria delle cardinalità dell'insieme delle classi d'equivalenza di ogni input ma può risultare necessaria per sistemi critici in cui bisogna privilegiare l'efficacia come per esempio quelli che gestiscono apparrecchi medici oppure voli. Il testing svolto in questo modo è esaustivo rispetto alle classi di equivalenza.

Capitolo 2: Automazione del Testing e Strumenti

Il testing manuale implica che un tester inserisca input predefiniti nel sistema utilizzando l'interfaccia utente, una console a riga di comando o il debugger. Dopo l'esecuzione dei test, il tester confronta gli output con l'oracolo.

Una possibile soluzione al testing manuale può essere quella di scrivere in ogni classe da testare un metodo “main” in cui si inseriscono comandi atti a stampare variabili critiche per poterle analizzare e successivamente procedere al debugging.

Uno degli svantaggi di tale approccio è che l'occhio umano può commettere errori nel fare questo confronto ed inoltre può richiedere una notevole quantità di tempo poiché è necessario, manualmente e ad ogni esecuzione, analizzare gli output alla ricerca di eventuali errori.

Un’altro problema è che, per distribuire il progetto, è necessario eliminare tutti i comandi per il testing inseriti nel codice poiché, in caso contrario, risulterebbe appesantito. Inoltre tale approccio può essere conveniente solo per programmi di piccole dimensioni mentre nel caso di grossi progetti diventerebbe tutto più difficile da gestire.

E’ ben noto come la validazione e la verifica di un programma rappresentino una fase molto importante nel ciclo di sviluppo di qualsiasi prodotto software ed è stato più volte stimato come il costo di tale fase si aggira facilmente attorno al 50% dell'intero costo di sviluppo del prodotto.

Per cercare di aumentare l'efficacia, tagliare i costi e ridurre i tempi di sviluppo sarebbe estremamente utile poter disporre di tools in grado di generare automaticamente grandi quantità di casi di test a partire dal codice o dalle specifiche del programa sotto

osservazione.

L’automazione del testing porta notevoli vantaggi:

- Tempo risparmiato nell’esecuzione dei test
- Affidabilità dei test poiché non c’è rischio di errore umano nell’esecuzione di questi ultimi
- E’ più facile garantire che lo stesso test sia eseguito nelle medesime condizioni ogni volta
- Riuso parziale dei test a seguito di modifiche nel codice
- Riduzione dei costi poiché non c’è bisogno di personale umano, cioè implica che è possibile privilegiare l’efficacia a discapito dell’efficienza

C’è la necessità di un approccio sistematico che possa separare il codice di test da quello della classe, che supporti la strutturazione dei casi di test in test suite e che fornisca un’output dei risultati del test separato dall’output dovuto all’esecuzione della classe.

2.1 EcFeed

EcFeed è un tool gratuito ideato per facilitare la creazione di casi di test e supporta la generazione automatica di dati di test tramite potenti algoritmi combinatoriali. Possiede un’interfaccia molto intuitiva che permette di progettare, modellare, analizzare ed eseguire test.

E’ disponibile sia come plugin di Eclipse che in versione standalone. La differenza è che quest’ultima non ha accesso al progetto Java nell’IDE di Eclipse quindi non può usare funzioni che hanno qualcosa in comune col codice sorgente. Noi in questa tesi tratteremo

la versione plugin che supporta anche l'integrazione con JUnit, essa permette di importare classi di test esistenti nel modello e viceversa: il modello può essere esportato in un progetto java Eclipse come stub di codice.

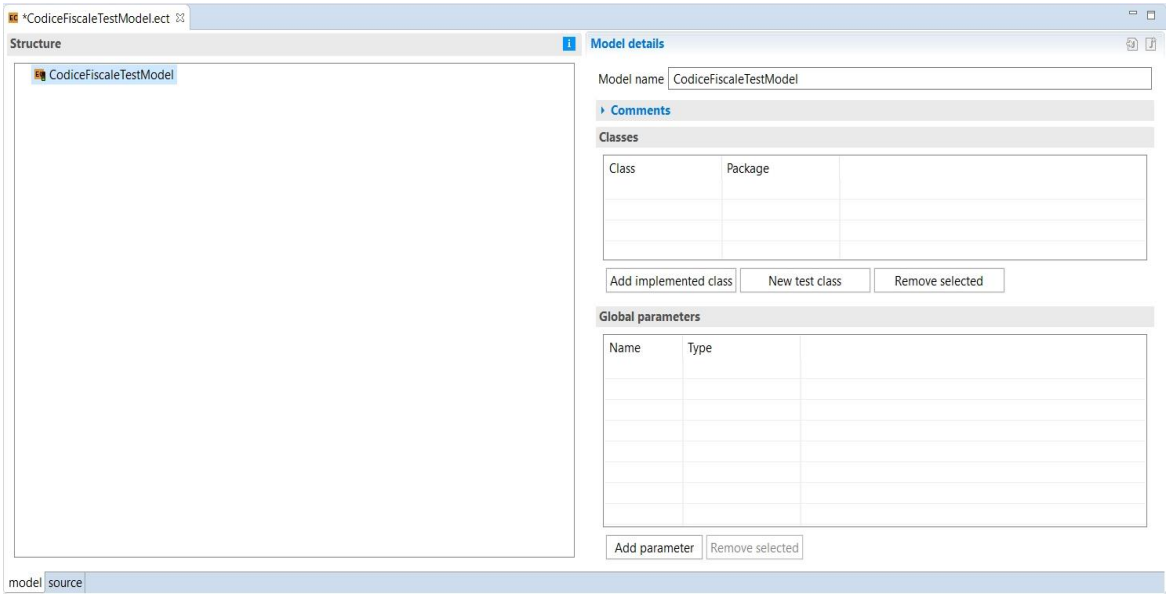
EcFeed è costruito intorno al concetto di classi di equivalenza, il prefisso "ec" nel nome del tool sta per "equivalence class" mentre la parte 'Feed' è un tributo al tool feed4junit da cui si è ispirato.

Questo tool usa tecniche avanzate di testing combinatoriale per generare casi di test che possono trovare i difetti più critici del software in maniera efficiente, può generare suite di test n-wise e supporta anche algoritmi per la adaptive random data generation.

2.1.1 Creazione e gestione di un modello di test

La prima cosa da fare per iniziare ad usarlo è creare un nuovo albero di classi di equivalenza andando su File->New->Other, esso avrà estensione .ect.

L'editor principale è diviso in due parti: a sinistra, nella pagina principale, si può navigare nell'albero del modello mentre a destra vi è la pagina dei dettagli dove il nodo selezionato può essere editato. Nel menu contestuale del nodo, attivabile col tasto destro del mouse, sono disponibili operazioni tipo il copia e incolla.



Possiamo aggiungere una classe di test nella sezione delle Classi nella pagina dei dettagli. Clickando su nell'angolo in alto a destra implementeremo tutte le classi nel progetto che al momento non sono implementate, noteremo che il semaforo rosso nell'editor del modello sarà diventato verde.

Nella pagina dei dettagli della classe possiamo aggiungere nuovi metodi nella sezione dei Metodi, successivamente andranno implementati premendo nuovamente su oppure aggiungendo l'implementazione alla classe di test manualmente. Nella nostra classe di test quindi avremo:

```
public void CodiceFiscaleTestMethod() {
        // TODO Auto-generated method stub
        System.out.println("CodiceFiscaleTestMethod()");
}
```

Possiamo aggiungere un metodo al nostro modello anche importandolo dal codice. Andiamo nella pagina dei dettagli della classe di test e nella sezione Altri metodi della Classe di Test troveremo la firma del metodo da importare, selezioniamolo e aggiungiamolo. Per poterli importare è necessario siano public, non restituiscano alcun valore e accettino solo tipi primitivi o enums come parametri.

Nella pagina dei dettagli del modello possiamo anche importare classi intere clickando sul tasto "Add implemented class" e selezionandola dal browser.

Per aggiungere dei parametri, gli argomenti del nostro metodo di test, bisogna andare nella pagina dei dettagli del testMethod() e nella sezione dei parametri clickare sul tasto "Add parameter". Qui si può cambiare il nome, il tipo e il flag "Expected" del parametro. Per usare parametri di tipo utente bisogna mettere come prefisso il nome del package Java. Ora ci sarà un nodo per ogni parametro del metodo, nella pagina dei dettagli del parametro possiamo aggiungere le choices tramite il tasto "Add choice" e quando ne aggiungiamo una il semaforo del parametro diventerà verde. Quando aggiungiamo un nuovo parametro al metodo esso avrà già delle choices di default che sono valori speciali e dipendono dal tipo: MIN_VALUE e MAX_VALUE.

Ora possiamo aggiungere qualche caso di test al nostro metodo che non è altro che una combinazione di choices rappresentanti i parametri dei metodi. Bisogna andare nella pagina dei dettagli del metodo e clickare su “Add test case” nella sezione dei Casi di Test. Comparirà una finestra di dialogo dove si potrà definire una combinazione di choices che saranno usati come parametri nel test case. Fatto questo nell’elenco troveremo un test case appartenente alla suite di default che sarà verde quindi eseguibile. Per eseguirlo basta selezionarlo e clickare su “Execute selected”.

2.1.2 Generazione di dati di test

Per generare una test suite bisogna andare nella pagina dei dettagli del metodo e nella sezione dei Casi di Test clickare su “Generate test suite”. Si aprirà una finestra dove la si può configurare: nella view in alto si possono selezionare le Constraints del nostro generatore, sotto possiamo selezionare le choices che saranno usate (ogni parametro deve averne almeno una), poi si possono settare il nome comune a tutti i casi di test generati e il tipo di generatore. Una volta confermate le scelte la suite sarà mostrata nella finestra “Test cases” e sarà eseguibile. I test singoli, di cui la test suite si compone, saranno visibili sono se in numero minore di 500.

Il tasto “Test online” sarà disponibile in alto nella pagina dei dettagli del metodo, questa funzionalità sarà usata nel caso in cui si abbia bisogno di eseguire i test cases direttamente quando sono generati senza salvarli.

I generatori sono classi che forniscono dati di test, ce ne sono di diversi tipi:

- Prodotto Cartesiano che genera tutte le combinazioni possibili delle choices configurate, quindi può risultare generato un numero enorme di test cases.
- N-wise che genera una suite che copre tutte le possibili n-ple dei nostri input. Nella finestra di dialogo bisogna impostare il parametro “N” a seconda se vogliamo consideri 2-ple, 3-ple ecc e il parametro “coverage” che definisce la percentuale di n-ple che vogliamo coprire. Tramite il tasto “Calculate coverage” nella sezione dei casi di test possiamo controllare la copertura al variare di N.

- Random che, come suggerisce il nome, genera un set di combinazioni casuali. I due parametri da settare solo “length” che indica il numero di casi di test che vogliamo generare e “duplicates” che è un flag che se non è spuntato fa in modo che non vengano generati due casi di test identici nella stessa suite.
- Adaptive random che è una generalizzazione di quello Random. Ad ogni passo genera un numero di test casuali e sceglie tra questi quelli più “lontani” (come distanza di Hamming) da quelli già generati. Rispetto al Random qui abbiamo anche il parametro “Candidate set size” che determina quanti casi di test candidati vengono generati ogni volta e il parametro “Depth” che è il numero dei casi di test recenti verso cui ogni candidato è confrontato per trovare il più lontano. Se lo settiamo a -1 i candidati verrano confrontati con tutti i casi di test generati finora.

2.1.3 Features avanzate

Le Constraints sono usate per definire dipendenze tra differenti parametri di input o tra parametri di input e output. Una constraint è espressa da due espressioni logiche: una premessa e una conseguenza. Possono essere usate per specificare:

- Quali combinazioni di valori di input devono essere incluse nel processo di generazione di una test suite in modo da filtrare quei valori di input che non hanno senso nel metodo testato. Verranno inclusi solo i casi di test per cui se la premessa è vera lo è anche la conseguenza, in caso contrario il caso di test è rimosso dalla suite risultante. Per esempio se si vuole evitare una certa combinazione di variabili di input basterà mettere nella premessa la combinazione stessa e settare la conseguenza a false. Invece se volessi bloccare il valore di una variabile di input basterebbe scrivere “variabile != valore ->false”.
- Assegnare un valore ad un parametro “expected”, in questo caso la conseguenza ha la forma di un’assegnazione

Per aggiungere una constraint ad un metodo basterà clickare su “Add constraint” nella finestra di dettaglio del metodo, cosi facendo si aggiungerà quella di default “true=>true” che risulta vera per ogni caso di test.

Se andiamo sul nodo della constraint creata possiamo editarla e cambiare il nome, la premessa e la conseguenza. Una volta create possiamo selezionarle, come visto prima, quando generiamo una test suite.

Un parametro di una funzione di test può essere impostato come expected spuntando la relativa voce nella pagina dei dettagli del parametro. Esso invece di essere utilizzato come input per l'oggetto testato è usato per verificare il suo output, tipicamente tramite l'istruzione assert. Per ogni caso di test possiamo impostare il valore del parametro expected con l'uso delle constraints, col valore di default o assegnandolo manualmente al test case. Essi non hanno choices (tranne per i tipi definiti dall'utente) quindi non modificheranno il numero di casi di test prodotto dai generatori.

Le choices possono essere combinate in gruppi per semplificare le constraints, basta creare una choice astratta e copiare e incollare le choices (oppure fare drag&drop) che si vuole facciano parte del gruppo. Cosi facendo vedremo nel modello le choices organizzate in gerarchie.

Le labels hanno una funzione simile alle choices astratte, si possono aggiungere un numero arbitrario di labels alle choices per poi usarle nelle constraints. Vengono utilizzate nel caso le choices siano già organizzate in gerarchie e non risulti conveniente modificare queste ultime. Per aggiungerle andare nella pagina dei dettagli della choice, clickare su "Add label" e successivamente darci un nome. Si possono assegnare labels anche alle choices astratte.

Nel caso si abbiano più funzioni che dipendono da argomenti simili i parametri globali risultano molto utili, essi possono avere visibilità all'interno dell'intero modello oppure solo della singola classe. Possono essere creati da zero oppure si può trasformare un parametro esistente in globale facendo il drag&drop nel nodo radice del modello coi tasti Ctrl and Shift premuti, cosi facendo il parametro sarà automaticamente linkato alla definizione globale del parametro. Per linkare altri parametri al globale bisogna andare alla pagina dei dettagli del parametro e spuntare la voce "Linked" per poi selezionare il parametro globale a cui linkarlo. Quindi il vantaggio è che se vogliamo aggiungere una nuova choice al nostro modello possiamo farlo in un unico posto e tutti i parametri linkati

alla definizione globale la useranno automaticamente.

EcFeed permette di esportare casi di test in files di testo. E' possibile esportare i singoli casi di test selezionati dal modello oppure esportarli direttamente dal generatore, senza salvarli, tramite il tasto "Export online" nella pagina dei dettagli del metodo. Si aprirà una finestra di dialogo con gli stessi campi visti nella generazione dei casi di test, in più avremo "Export target file" che serve per selezionare il file che verrà creato e potremo scegliere tra tre templates: CSV,XML e Gherkin. Di default EcFeed genera un file csv usando tutti i parametri del metodo di test, mette i nome dei parametri nella prima riga e i valori per i casi di test generati in quelle successive. Se abbiamo necessità di esportare i dati in un modi differenti basta clickare su tasto "Advanced" e verremo reindirizzati su una finestra che ci permetterà di definire un template. Quest'ultimo consiste in tre sezioni: Header, TestCase e Footer dove solo il secondo è obbligatorio. Per esportare solo casi di test selezionati dal modello: basta andare nella sezione Test Cases, scegliere quelli da esportare e clickare su "Export selected".

2.1.4 Integrazione con JUnit

E' possibile integrare EcFeed con JUnit, tale feature risulta particolarmente utile per il regression testing. EcFeed fornisce JUnit con delle classi runner che sono capaci di analizzare il modello ed eseguire tests. Bisogna scaricare ed installare i jars ecfeed e xom che devono poi essere aggiunti alla Java Build Path del progetto insieme alla libreria di JUnit

I test generati, però, non saranno indipendenti da EcFeedRunner a meno che non poniamo come output dei test stessi il codice di test indipendenti.

Per eseguire i test di EcFeed con JUnit è necessario usare le annotazioni @RunWith, @EcModel e @Test nell'implementazione del metodo. L'annotazione @RunWith(Runner.class) definisce il runner responsabile per eseguire i test definiti in una classe di test. Uno dei runner che è possibile usare è lo StaticRunner la cui classe è definita nella libreria ecfeeed.jar. Esso è responsabile dell'esecuzione dei casi di test forniti nel modello. Un' altro runner che è possibile usare è l'OnlineRunner che permette di generare

test online durante l'esecuzione, quindi non verranno salvati nel modello. Tale approccio è vantaggioso al crescere delle dimensioni del modello ma rende impossibile modificare i casi di test.

Il runner carica il modello dalla locazione fornita dall'annotazione @EcModel("path/to/model/file.ect") dove la path può essere assoluta o locale. I runner forniti da EcFeed sono responsabili per eseguire i casi di test definiti dai metodi annotati con @Test. Essi prima controllano se il metodo ha dei parametri: nel caso affermativo cercano di trovare il corrispondente metodo nel modello ed eseguono i casi di test del modello, nel caso non li abbia passano l'esecuzione al runner standard di jUnit quindi saranno lo stesso eseguiti anche se non sono modellati.

Una volta generata una test suite per avviarla con jUnit basterà premere Alt+Shift+x oppure andare su Run->Run dal menu di eclipse.

Un'altra annotazione che è possibile usare è @TestSuites({"Test suite 1", "Test suite 2"}) che serve a indicare quali suite di test saranno eseguite dal metodo indicato. Di default saranno eseguiti tutti i casi di test definiti per un dato metod. Il parametro dell'annotazione è una stringa che contiene il nome della suite di test da eseguire, può prendere come parametro anche un array di stringhe per definire più suite di test. L'annotazione può essere usata sulla classe o su un metodo di test, nell'ultimo caso le suite di test definite sovrascriveranno quelle definite globalmente.

L'annotazione @Generator(Generator.class) è necessario sia definita per le classi che usano l'OnlineRunner o altri runner che generano dati di test a runtime. Essa serve per indicare al runner quale generatore usare, quest'ultimo genera nuovi casi di test durante l'esecuzione e li fornisce al runner finchè tutti i test sono stati eseguiti. Il più semplice da utilizzare è il CartesianProductGenerator perché non richiede parametri, altri come il RandomGenerator ne prevedono e possiamo fornirli tramite l'annotazione @GeneratorParameter(name="name", value="value"). Essi sono, come già visto, length e duplicates dove quest'ultimo è opzionale e settato di default a false. Nel caso vogliamo fornirli entrambi non è possibile usare due annotazioni @GeneratorParameter ma dobbiamo usare @GeneratorParameterNames({"par1", "par2", "par3"}) e

@GeneratorParameterValues({"val3", "val2", "val3"}). La prima definisce un array coi nomi dei parametri e la seconda un array dei rispettivi valori che essi assumono, ovviamente devono avere entrambi la stessa dimensione.

I generatori possono essere configurati in modo da usare constraints fornite dal modello tramite l'annotazione @Constraints({"name1", "name2"}). I parametri sono le singole constraints che verranno usate dal generatore. Si possono usare anche due speciali valori Constraints.All e Constraints.NONE: il primo indica che verranno usate tutte le constraints mentre il secondo dice al generatore di ignorarle tutte.

Anche le annotazioni @Generator, @GeneratorParameter e @Contraints, come già visto per @TestSuites, possono essere definite sia globalmente (a livello di classe) che localmente per un particolare metodo a seconda di dove si posiziona l'annotazione.

2.1.5 Testing di pagine web

EcFeed permette di fare test alle pagine web senza scrivere alcun codice. Esso apre un browser sulla pagina web da testare e sollecita gli elementi della pagina con azioni che sono mappati a parametri di metodi. Per usarlo c'è bisogno di aver installato, oltre ad un browser a scelta tra cui Chrome, i drivers di selenium per il browser scelto.

La prima cosa da fare è aggiungere una classe di test e un metodo al modello, successivamente nella pagina dei dettagli del metodo selezionare Web runner dal menu a tendina del Runner e selezionare Chrome nelle proprietà del Web runner. Nel campo Web driver digitare la path dove è stato decompresso il driver Selenium di Chrome e nel campo Start URL l'URL della pagina web da testare.

Se si preme il tasto Test online EcFeed aprirà Chrome sulla pagina richiesta, per non farla chiudere immediatamente è necessario aggiungere un parametro numerico al metodo e assegnargli come tipo "Delay[s]" nella finestra dei dettagli del parametro. Successivamente aggiungiamo una choice a quest'ultimo che sarà il tempo in secondi in cui la nostra pagina sarà visibile prima di chiudersi.

Per testare il titolo di una pagina web aggiungiamo al metodo un parametro di tipo stringa e nella pagina di dettaglio gli diamo un nome, lo settiamo ad Expected e come valore di

default inseriamo il titolo che ci aspettiamo la pagina web abbia. Nelle sezione delle proprietà del Web runner selezioniamo come tipo dell'elemento Text e Identified by Id dove l'id è quello associato al titolo della pagina web che stiamo testando. Cosi' facendo abbiamo indicato ad EcFeed che il parametro di tipo stringa creato è mappato ad un elemento di testo nella pagina identificato dall'id inserito.

Allo stesso modo di quanto visto per il titolo possiamo manipolare gli elementi di pagine web aggiungendo nuovi parametri. I tipi degli elementi tra cui possiamo scegliere sono:

- Unmapped. Il parametro non è connesso ad alcun elemento della pagina web
- Text. Gli elementi di testo si trovano tra vari tags html: p, div, td, tr ecc.
- Select. Menù a tendina html
- Radio. Tasto radio
- Checkbox. Tasto checkbox
- Button. Input html di tipo tasto
- Page element. Tipo universale a cui corrisponde qualunque elemento in una pagina web
- Delay. Un parametro che serve a mettere in pausa il test per un tempo specificato in secondi
- Page URL. URL di una pagina web
- Browser. Il tipo di browser che esegue il test

Il set di tipi degli elementi che è possibile usare dipende dal tipo scelto per il parametro. Per esempio Button e Checkbox sono disponibili per i tipi booleani, Delay per tipi numerici mentre Select, Radio, Page URL e Browser per tipi stringa.

Una volta che abbiamo identificato un elemento web possiamo eseguire delle azioni specifiche su di esso. Il set di azioni disponibili dipende dal tipo dell'elemento e se il parametro è Expected. Per esempio se abbiamo un elemento di tipo Text, nel caso il parametro sia expected possiamo usare l'azione "send keys" per scriverci dentro (a patto che sia definito col tag di input nella pagina web) e nel caso non lo sia l'azione "check value". Per Checkbox e Button l'unica azione disponibile è click, in questo caso il campo Action nella pagina di dettaglio del metodo rimarrà disabilitato poiché possiamo scegliere

solo l’unica.

Per far partire il test bisogna andare nella pagina dei dettagli del metodo e clickare sul tasto Test online. Si aprirà la pagina web dal browser selezionato e i campi saranno riempiti coi valori impostati in precedenza.

Se i campi impostati prevedono di trovarci in una nuova pagina web, ad esempio dopo aver simulato la pressione di un tasto, possiamo verificare se l’indirizzo è lo stesso di quello che ci aspettiamo. Per farlo dobbiamo aggiungere un nuovo parametro expected di tipo stringa, settare il tipo a Page URL e mettere come valore di default l’URL dove ci aspettiamo di essere stati reindirizzati.

Possiamo definire dei parametri expected che controllano valori creati dinamicamente durante l’esecuzione dei test come per esempio una label. Se il valore non corrisponde a quello expected il report del test ci dirà che vi è stato un problema.

Nella pagina di dettagli del parametro possiamo settarlo ad Optional tramite il corrispondente checkbox. Cosi’ facendo l’elemento che è mappato col parametro sarà controllato solo se esso esiste nella pagina (non è nascosto o disabilitato), in caso contrario sarà ignorato e il caso di test non fallirà.

Capitolo 3: Valutazione dell'efficacia dei Test

Per poter valutare oggettivamente e sperimentalmente quanto le tecniche black-box viste in precedenza siano state efficaci dobbiamo necessariamente spostarci in ambito white-box, in caso contrario potremmo solamente contare i malfunzionamenti trovati ma, siccome non sappiamo quanti ce ne possano essere, questa misura non ci risulta molto utile per valutare l'efficacia.

Lo metodica più utile per valutare l'efficacia dei test è quella che verifica la copertura del codice: maggiore è la copertura del codice e più alta sarà la probabilità di aver scoperto un numero alto di malfunzionamenti e, di conseguenza, avremo un numero minore di malfunzionamenti residui quindi software di qualità migliore. La copertura usa una scala che va dallo 0% al 100% dove il massimo si raggiunge quando i test esercitano in maniera completa il software. Indagini statistiche hanno evidenziato che la soglia ottimale si raggiunge con una copertura vicina all'85%, in generale bisogna considerare il trade-off tra efficacia del test e costi intesi sia in termini economici che temporali.

Esistono vari criteri per la copertura:

- Statement coverage (copertura delle istruzioni). Consiste nel trovare un insieme di ingressi che garantisca l'esecuzione di ogni istruzione presente nel codice almeno una volta.
- Branch coverage (copertura delle decisioni). Parte dal presupposto che è necessario esaminare ogni ramo del diagramma di flusso sia per il valore di verità che di falsità. Potrebbe sembrare sufficiente per verificare tutte le decisione ma esistono due eccezioni: quando il programma non presenta alcuna decisione e quando una

sezione di esso presenta più punti di ingresso quindi viene una parte viene eseguita opzionalmente.

- Condition coverage (copertura delle condizioni). Si devono scrivere un insieme di casi di test tali che ciascuna condizione elementare assuma tutti i possibili valori almeno una volta. Questo criterio, come quello del branch coverage, andrebbe affiancato allo statement coverage per verificare anche le possibili eccezioni che sorgono durante l'esecuzione dei test.
- Copertura delle condizioni e decisioni. Bisogna cercare un insieme di casi di test tali da garantire che ogni condizione assuma il valore di verità e falsità almeno una volta e che ogni decisione sia percorsa almeno una volta sia per il ramo vero che per quello falso. Tuttavia va considerato che un elaboratore non è in grado di verificare condizioni multiple senza spezzarle in più condizioni singole su cui lavorare.
- Copertura delle condizioni multiple. Bisogna strutturare i casi di test in modo che essi prevedano tutti i possibili valori di ciascuna combinazione di condizioni presenti in una decisione.

3.1 EclEmma

EclEmma è un tool freeware che ci permette di analizzare la copertura di codice java, è disponibile sotto forma di plugin per Eclipse installabile dal marketplace. Non richiede modifiche del progetto o effettuare altri tipi di setup.

EclEmma aggiunge al workbench una nuova modalità di lancio chiamata "Coverage" che funziona esattamente come le esistenti Run e Debug e può essere attivata dal menu Run oppure nella toolbar.

Le informazioni sulla copertura saranno automaticamente disponibili nel workbench di eclipse dopo la terminazione del programma oppure su richiesta. Comparirà una nuova scheda nel pannello inferiore dell'interfaccia di eclipse che riepiloga la copertura della sessione corrente, essa mostrerà un elenco degli elementi analizzati sotto forma di gerarchia dando informazioni sulla percentuale di codice coperta e linee totali. E' possibile

ordinarli in ordine discendente o ascendente clickando sulla rispettiva colonna del parametro a cui siamo interessati.
I risultati della sessione saranno anche direttamente visibili in maniera grafica nell'editor del codice tramite le annotazioni: verranno evidenziate le linee di codice in colori differenti. I colori di default, che possono essere personalizzati nelle opzioni, sono:

- Verde per le linee di codice coperte pienamente
- Giallo per quelle coperte parzialmente in cui qualche istruzione o ramo è stato mancato
- Rosso per quelle che non sono state per niente eseguite

Una sessione di coverage sono le informazioni sulla copertura del codice di una particolare esecuzione del programma ed è automaticamente cancellata quando si chiude il workbench. Se il test totale consiste nell'esecuzione di più test ci saranno più sessioni, è possibile fare il merge tra di loro per creare una singola sessione da analizzare.
Nonostante EclEmma sia stato progettato principalmente per l'esecuzione e l'analisi dei test nel workbench di eclipse fornisce la possibilità di fare import/export, in particolare possiamo esportare le statistiche raccolte durante la sessione in HTML, XML o CSV andando nel menu file-> export -> Run/Debug -> Coverage Session.
EclEmma permette anche di verificare la copertura del codice relativo ai casi di test JUnit facendo Run -> Coverage As -> JUnit test.
Siccome dalle versione 2.0 EclEmma è basato su JaCoCo, anch'essa una libreria per la copertura, riprende le sue stesse definizioni per il conteggio.
Essi grazie all'uso di contatori derivati dalle informazioni contenuti nelle classi java stesse (il byte code e le informazioni di debug) riescono ad offrire un'instrumentazione dell'applicazione on-the-fly, anche quando il codice sorgente non è disponibile. Ciò semplifica molto l'analisi della copertura del codice.
Nel riepilogo possono essere selezionate differenti modalità di conteggio:

- Istruzioni. Conta singole istruzioni del byte code java che sono state eseguite, esse sono la più piccola unità che è possibile contare. Questa metrica è indipendente

dalla formattazione del codice sorgente.

- Rami. Conta il numero totale di rami, derivati dalle istruzioni if e switch, nei metodi e determina il numero di rami eseguiti o mancati. La gestione delle eccezioni non viene considerata come ramo. Se la classe è stata compilata con le informazioni di debug i punti di decisione verranno evidenziati come descritto in precedenza.
- Linee. Una linea di codice è considerata eseguita quando almeno una istruzione assegnata a questa linea è stata eseguita. Il conteggio delle linee è diverso da quello delle istruzioni perché una singola linea tipicamente viene compilata generando più istruzioni in byte code. Questa metrica di conteggio è disponibile solo per le classi compitate con le informazioni di debug. In molti casi le informazioni raccolte possono essere evidenziate coi colori nel codice ma ci sono delle limitazioni: non tutti i costrutti java possono essere compilati direttamente nel byte code corrispondente, in questi casi il compilatore crea del codice sintetico che qualche volta falsa i risultati di copertura.
- Complessità ciclomatica. Viene calcolata per ogni metodo non abstract e riepilogata per ogni classe, package e gruppo. La complessità ciclomatica, per definizione, è il minimo numero di percorsi che può, in combinazione lineare, generare tutti i possibili percorsi attraverso un metodo. EclEmma la calcola come: $v(G) = B – D + 1$ dove B rappresenta il numero di rami e D quello dei punti di decisione.
- Metodi. Un metodo viene conteggiato come eseguito se almeno una istruzione all'interno di esse è stata eseguita. Anche le chiamate al costruttore e inizializzatori statici sono contati come metodi.
- Classi. Una classe viene conteggiata come eseguita quando almeno un metodo è stato eseguito.

Capitolo 4: Esempio Pratico

Ora sarà presentato un esempio pratico in cui verranno applicate le tecniche e strumenti di cui si è parlato in precedenza. Testeremo il funzionamento di un programma che esegue il calcolo del codice fiscale a partire dalle generalità di una persona ovvero nome, cognome, sesso, data e città di nascita.

Il primo passo da effettuare è la ricerca delle classi di equivalenza per ognuno degli input, esse potrebbero essere :

- Nome : {nome ∈ nomi validi}, {nomi ∉ nomi validi}
- Cognome : {cognome ∈ cognomi validi}, {cognome ∉ cognomi validi}
- Città : {città ∈ città valide}, {città ∉ città valide}
- Giorno : {giorno<=0}, {1<=giorno<=28}, {giorno=29}, {giorno=30},{giorno=31}, {giorno>31}
- Mese : {mese<=0}, {1< =mese<=12}, {mese>12}
- Anno : {anno<=1853}, {1854<=anno<=2018}, {anno>2018}
- Sesso : {sesso=M}, {sesso=F}, {sesso≠M ∪ sesso≠F}

Per quanto riguarda i primi 2 input di tipo stringa, il nome e il cognome, possiamo considerare non validi quelli che contengono caratteri speciali come per esempio #,@ etc quindi all'interno delle classi di equivalenza possiamo per esempio prendere i seguenti valori: {nome=Giorgio}, {nome= #iorgio} e {cognome=Rossi}, {cognome= @ossi}.
Riguardo la città possiamo supporre (ricordiamo che per ora siamo in modalità black-box)

che il programma abbia una lista delle città valide con cui confronterà quella data in input, quindi prendiamo: {citta=Roma}, {citta=Plutolandia}.

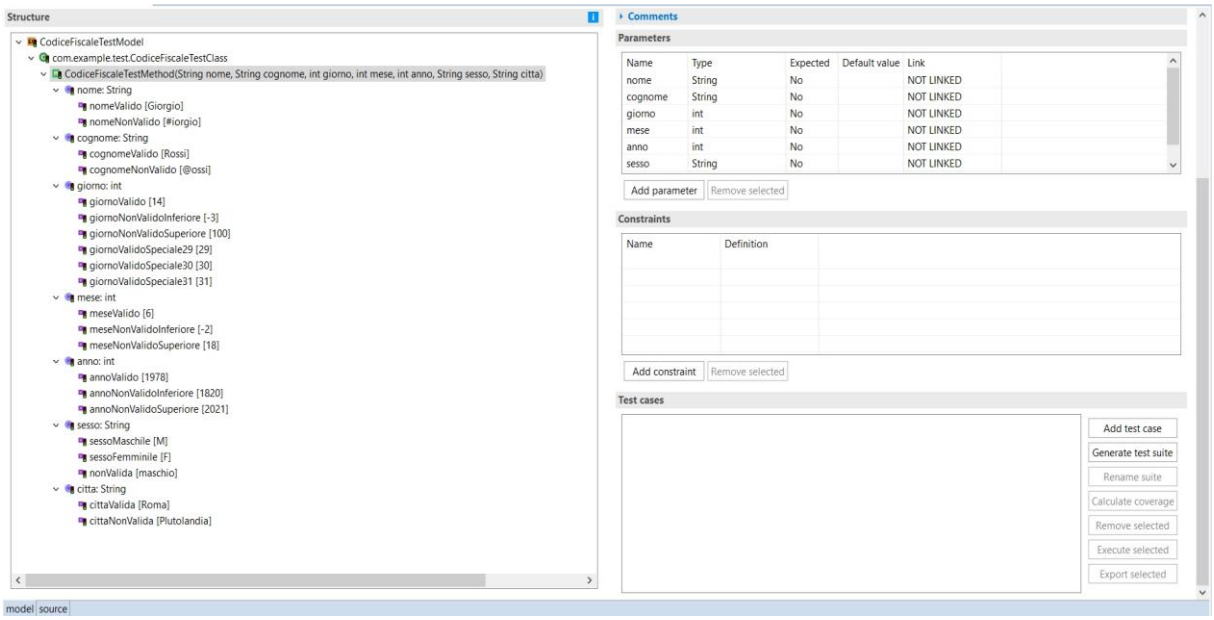
Per la data di nascita i giorni 29, 30 e 31 sono stati isolati in classi singole poichè, essendo giorni non presenti in tutti i mesi, potrebbero essere trattati in maniera speciale dal programma, sceglieremo : {giorno=-3}, {giorno=14}, {giorno=29}, {giorno=30}, {giorno=31}, {giorno=100}. Per il mese: {mese=-2}, {mese=6}. {mese=18}.

Per l'anno si è tenuto conto che l'età massima di vita documentata è circa 120 anni e che il codice fiscale è entrato in vigore nel 1973, inoltre una persona non può essere nata nel futuro. Sceglieremo: {anno=1820}, {anno=1978}, {anno=2021}.

Per il sesso è conveniente prendere ognuno dei due elelementi facenti parte dell'insieme discreto poichè ci dovrebbero essere differenze sostanziali nel modo in cui sono trattati dal programma, scegliamo: {sesso=M},{sesso=F},{sesso=maschio}.

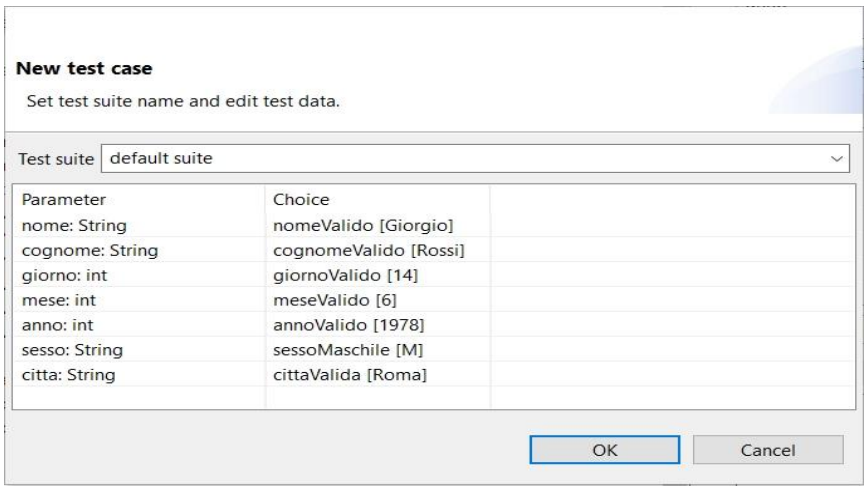
Da notare che le classi non valide dovute alla non appartenenza del valore al tipo sono state omesse poiché non necessarie nell'ambito di test di unità: il compilatore genererebbe errore durante la chiamata ad una funzione con un tipo errato.

Implementando il modello all'interno di EcFeed avremo questo risultato :

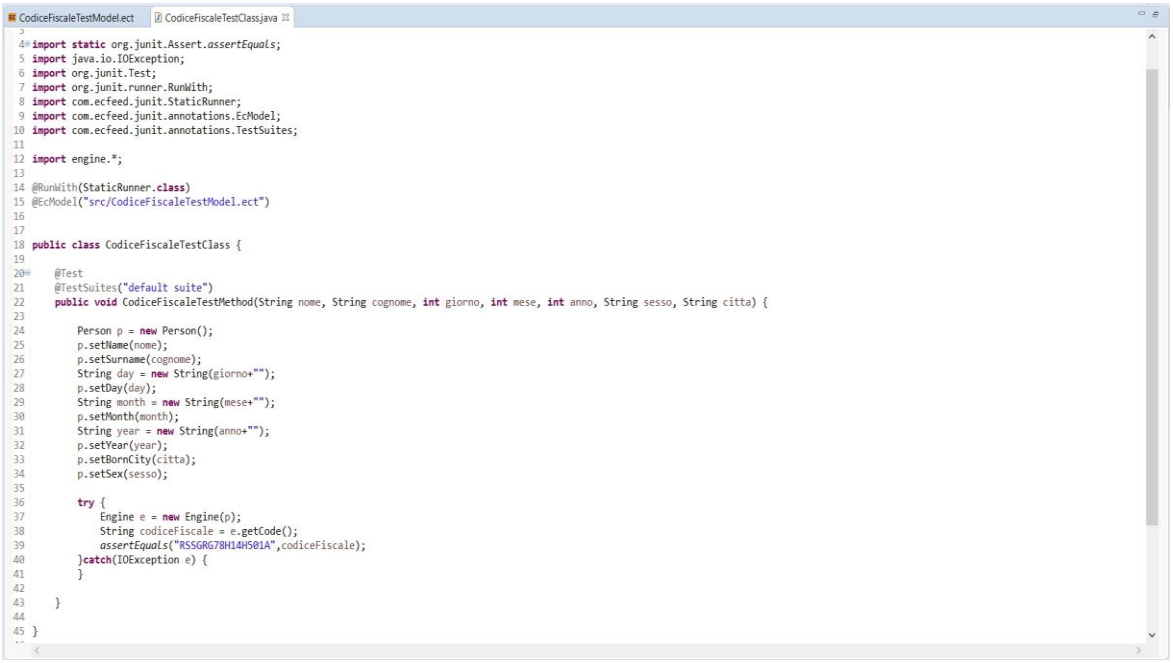


Non abbiamo avuto bisogno di implementare alcuna constraint poichè non ci sono combinazioni che non ha senso testare.

Proviamo a generare un singolo caso di test composto da soli classi valide:

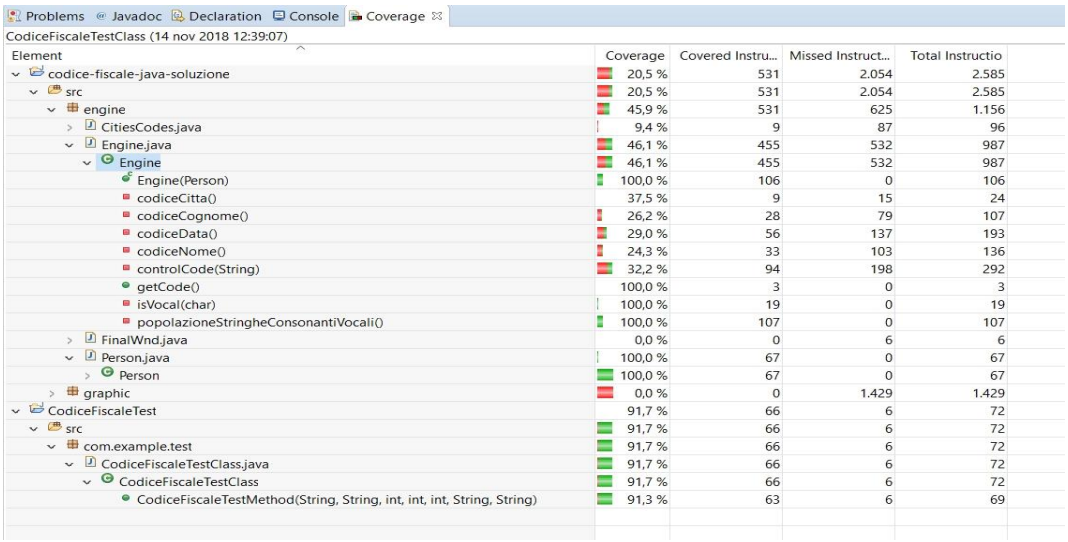


Poi andiamo a modificare il codice del metodo di test aggiungendo le annotazioni in modo da eseguirlo con JUnit. Importiamo la classe che vogliamo testare (in questo caso è chiamata “engine”) e le classi necessarie per JUnit ed EcFeed, instanziamo un oggetto Persona a cui passiamo i parametri dati in input al metodo di test ed infine passiamo l’oggetto al costruttore della classe che ha il compito di calcolare e conservare il codice fiscale generato. L’istruzione “assertEquals” verifica se il codice fiscale calcolato è uguale a quello atteso per questo specifico caso di test, in caso contrario JUnit genera una failure.


CodiceFiscaleTestModel.ect | CodiceFiscaleTestClass.java

```
import static org.junit.Assert.assertEquals;
import java.io.IOException;
import org.junit.Test;
import org.junit.runner.RunWith;
import com.ecfeed.junit.StaticRunner;
import com.ecfeed.junit.annotations.EcModel;
import com.ecfeed.junit.annotations.TestSuites;

import engine.*;

@RunWith(StaticRunner.class)
@EcModel("src/CodiceFiscaleTestModel.ect")


public class CodiceFiscaleTestClass {

    @Test
    @TestSuites("default suite")
    public void CodiceFiscaleTestMethod(String nome, String cognome, int giorno, int mese, int anno, String sesso, String citta) {

        Person p = new Person();
        p.setName(nome);
        p.setSurname(cognome);
        String day = new String(giorno+"");
        p.setDay(day);
        String month = new String(mese+"");
        p.setMonth(month);
        String year = new String(anno+"");
        p.setYear(year);
        p.setBornCity(citta);
        p.setSex(sesso);

        try {
            Engine e = new Engine(p);
            String codiceFiscale = e.getCode();
            assertEquals("RSSGRG78H14H501A",codiceFiscale);
        }catch(IOException e) {
        }

    }

}
```

Facciamo partire il test con JUnit e verifichiamo la copertura tramite EclEmma:

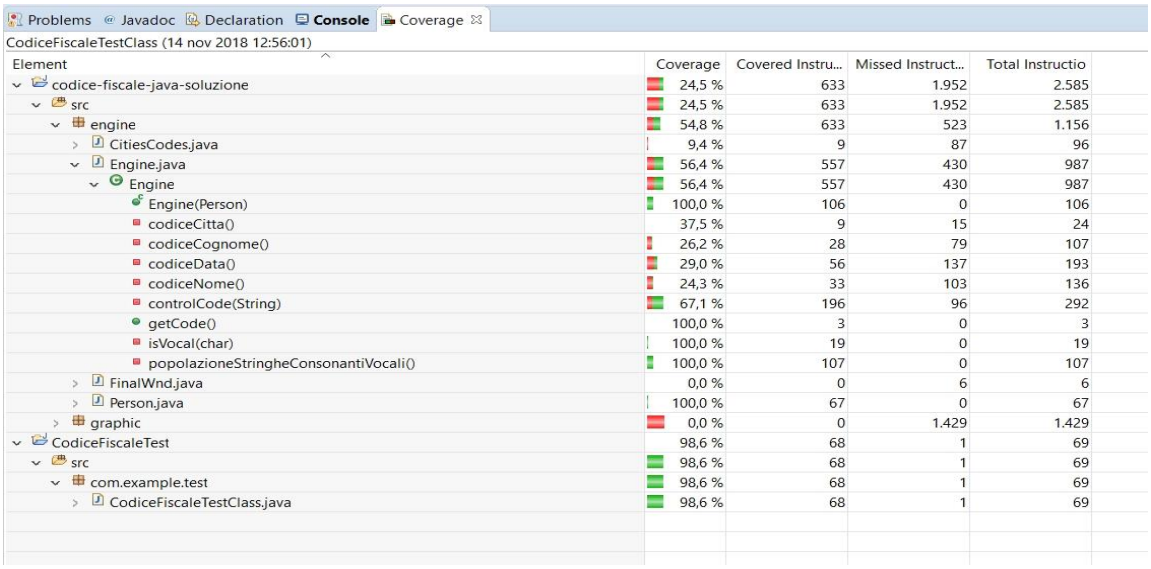


Element	Coverage	Covered Instru...	Missed Instruct...	Total Instructio
codice-fiscale-java-soluzione	20,5 %	531	2.054	2.585
src	20,5 %	531	2.054	2.585
engine	45,9 %	531	625	1.156
CitiesCodes.java	9,4 %	9	87	96
Engine.java	46,1 %	455	532	987
Engine	46,1 %	455	532	987
Engine(Person)	100,0 %	106	0	106
codiceCitta()	37,5 %	9	15	24
codiceCognome()	26,2 %	28	79	107
codiceData()	29,0 %	56	137	193
codiceNome()	24,3 %	33	103	136
controlCode(String)	32,2 %	94	198	292
getCode()	100,0 %	3	0	3
isVocal(char)	100,0 %	19	0	19
popolazioneStringheConsonantiVocali()	100,0 %	107	0	107
FinalWnd.java	0,0 %	0	6	6
Person.java	100,0 %	67	0	67
Person	100,0 %	67	0	67
graphic	0,0 %	0	1.429	1.429
CodiceFiscaleTest	91,7 %	66	6	72
src	91,7 %	66	6	72
com.example.test	91,7 %	66	6	72
CodiceFiscaleTestClass.java	91,7 %	66	6	72
CodiceFiscaleTestClass	91,7 %	66	6	72
CodiceFiscaleTestMethod(String, String, int, int, int, String, String)	91,3 %	63	6	69

Come possiamo notare abbiamo coperto solo il 46,1% della classe Engine tramite questo singolo test case.

Se proviamo a generare una test suite col generatore cartesiano avremo ben 1296 combinazioni (2*2*6*3*3*3*2=1296) ed EcFeed ci avvertirà che lo strumento potrebbe rallentare in presenza di un numero così elevato di test case. Nel codice del metodo di test togliamo l'assert e stampiamo a video i codici fiscali trovati.

Verifichiamo che la copertura è aumentata al 56,4%.

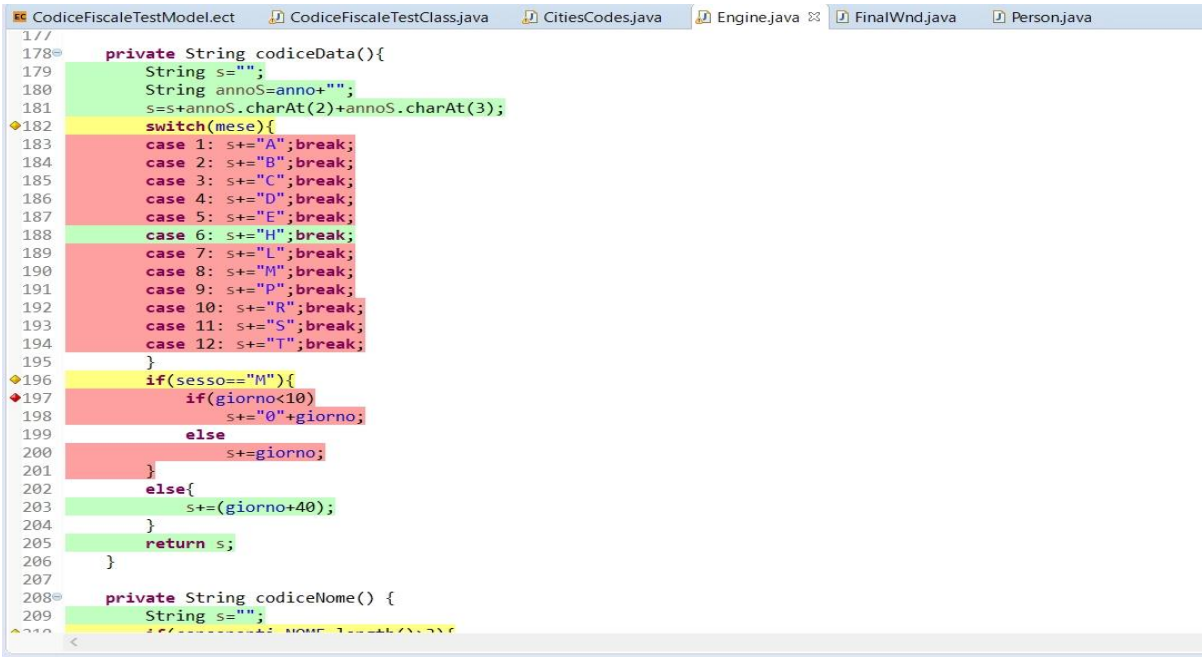


Element	Coverage	Covered Instru...	Missed Instruct...	Total Instructio
codice-fiscale-java-soluzione	24,5 %	633	1.952	2.585
src	24,5 %	633	1.952	2.585
engine	54,8 %	633	523	1.156
CitiesCodes.java	9,4 %	9	87	96
Engine.java	56,4 %	557	430	987
Engine	56,4 %	557	430	987
Engine(Person)	100,0 %	106	0	106
codiceCitta()	37,5 %	9	15	24
codiceCognome()	26,2 %	28	79	107
codiceData()	29,0 %	56	137	193
codiceNome()	24,3 %	33	103	136
controlCode(String)	67,1 %	196	96	292
getCode()	100,0 %	3	0	3
isVocal(char)	100,0 %	19	0	19
popolazioneStringheConsonantiVocali()	100,0 %	107	0	107
FinalWnd.java	0,0 %	0	6	6
Person.java	100,0 %	67	0	67
graphic	0,0 %	0	1.429	1.429
CodiceFiscaleTest	98,6 %	68	1	69
src	98,6 %	68	1	69
com.example.test	98,6 %	68	1	69
CodiceFiscaleTestClass.java	98,6 %	68	1	69

Usando un generatore 2-wise il numero di casi di test generato sarà 22 e raggiungiamo una copertura del 52,2% quindi non molto differente da quella ottenuta col cartesiano.

Possiamo notare che il 2-wise rappresenta un compromesso molto migliore poiché eseguendo l'1,69% dei casi di test del generatore cartesiano raggiungiamo il 92,5% della copertura raggiunta da quest'ultimo!

Ora passiamo alla modalità white-box e ispezioniamo la classe per vedere le parti che sono/non sono state coperte:



In particolare notiamo che in questo switch, appartenente alla funzione che calcola la parte del codice relativa al mese, abbiamo coperto un unico ramo: quello del mese 6 che abbiamo scelto per la classe valida del mese. Ciò significa che la scelta iniziale per le classi di equivalenza del mese non è stata soddisfacente poiché abbiamo verificato che i mesi da 1 a 12 non sono tutti trattati in maniera equivalente dal programma, per coprire tutti i rami bisognerebbe assegnare ciscun mese di questo intervallo ad una classe di equivalenza distinta.

Questo è proprio uno dei limiti concettuali del testing black box con classi di equivalenza: non avendo accesso al codice non possiamo essere sicuri che la scelta delle classi di equivalenza sia stata soddisfacente.

Conclusioni

Questa tesi ha avuto come scopo quello descrivere un approccio al testing black-box basato su classi di equivalenza attraverso lo strumento EcFeed che, usato in congiunzione con JUnit, permette di automatizzare buona parte del processo di testing. L'automazione del testing è molto importante per gli aspetti descritti in precedenza, in particolar modo per avere risultati dei test il più possibile esenti da errori e per ridurre i costi della fase di testing che da sola, ricordiamo, impiega spesso circa il 50% del budget totale di un progetto.

EcFeed è un tool relativamente giovane e fino ad un anno fa non aveva neanche un sito web con la documentazione, il suo sviluppo sta procedendo abbastanza speditamente e sono disponibili frequentemente nuove beta. L'installazione di queste ultime si è rivelata necessaria per problemi di interazione con le recenti versioni di Eclipse.

In futuro è auspicabile che il numero di strumenti per l'automazione del testing aumenti ed essi siano progettati in modo da essere sempre più indipendenti da specifiche tecnologie o strumenti.

In particolar modo questi tools potrebbero migliorare nel testing di programmi concorrenti, per esempio JUnit non è garantito che riesca a rilevare problemi dovuti alla concorrenza come i deadlock e le race conditions.

Bibliografia

[1] Autori, Titolo dell'articolo, Nome della rivista, volume, pagine, Data

[2] Autori, Titolo del Libro, Editore, Anno, pagine.

[3] Autori, Titolo dell'articolo, Nome della conferenza di cui è agli atti, editore, anno, pagine.

[4] Nome del sito web, indirizzo http, ultima data in cui è stato acceduto