Integrazione di test di unità con Combinatorial Testing

Jacopo Federici

1025458

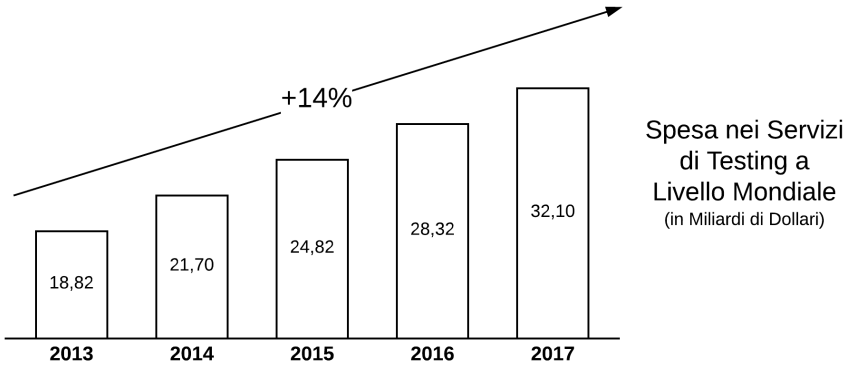
20 Marzo 2017

Indice

1 Introduzione

Nell'industria dell'informatica il testing del software sta assumendo sempre più peso. Una analisi di mercato condotta dalla Technavio [1] prevede una stabile crescita a livello mondiale del settore del testing del software di circa il 14% entro il 2021. Uno dei fattori chiave responsabili della crescita di questo mercato è l'aumento di servizi di test e servizi di automazione di test da parte di aziende specializzate.

I servizi che testano software per conto terzi sono spesso organizzati con metodologia Agile riuscendo a fornire un time-to-market più rapido con una maggiore efficienza. Questo aspetto rende molto vantaggioso ai produttori di software terziarizzare il test del prodotto sviluppato.



Il testing di applicazioni per il settore mobile è un caso interessante per analizzare il livello di sviluppo dell'automazione di test ed uno dei tipi di rapporti che si stanno instaurando tra coloro che producono software e coloro che lo testano.

In particolari contesti il testing del software non può che essere condotto dallo stesso produttore. In altri invece, solo aziende specializzate riescono a fornire un ambiente di test abbastanza ampio da rendere gli stessi test sul prodotto software validi ed affidabili.

Riprendendo l'esempio dello sviluppo di app mobile, sono ormai diversi i servizi che offrono un ecosistema con i quali lo sviluppatore può analizzare il comportamento della propria app su diversi smartphone, con diverse versioni del sistema operativo, al variare di molteplici parametri.
Un esaustivo testing interno, in questo settore, risulterebbe costoso e poco efficiente.

Dal punto di vista geografico, gli Stati Uniti rappresentano la maggior quota di mercato e si prevede che continueranno a dominare questo mercato nei prossimi anni. I clienti del settore bancario e delle telecomunicazioni formano una importante parte della domanda.
I test sono principalmente cloud-based, sfruttando le grandi capacità di calcolo che spesso sono necessarie con alcune tecniche di testing. Questo consentirà ad altri attori di entrare nei prossimi anni in questo mercato.
Un aspetto importante che viene mostrato in questo rapporto è la forte trasversalità del testing del software. Moltissimi prodotti e servizi di diversa natura sono diventati vasti ed articolati e la difficoltà di analizzare il loro corretto funzionamento è aumentata di molto.

2 Stato dell'arte del testing

2.1 JUnit

JUnit è un framework per test di unità per il linguaggio di programmazione Java. Fa parte della famiglia di framework di unit testing conosciuti come xUnit. È stato creato da Kent Beck e Erich Gamma, personaggi di spicco rispettivamente nel settore delle metodologie di sviluppo software (XP) e nel settore dei design pattern (metodologie Agile).

L'ultima versione rilasciata è la numero 4 che ha visto un forte cambiamento strutturale rispetto alle precedenti.

Lo scopo di JUnit, ed in generale dei framework per il test di unità, è quello di fornire degli efficienti strumenti per verificare il corretto funzionamento di singole unità di codice in determinate condizioni.

Nei linguaggi di programmazione orientata agli oggetti come Java, l'unità di codice è rappresentata da uno o più metodi che implementano una singola funzionalità.

I test di unità sono composti da Test Case, ovvero i test sulla singola unità di codice, e da Test Suite, un insieme di Test Case che verificano più funzionalità correlate fra loro.

Consideriamo ad esempio un metodo che sviluppa la funzionalità di conversione di un numero positivo passato in input in un numero negativo restituito in output. Un test Case di questo metodo si occuperà di verificare che, al passaggio del numero positivo, ad esempio 2, il metodo restituisca -2. Se la verifica fallisce il codice implementato nel metodo è evidentemente da correggere.

La versione 4 di Junit, con l'introduzione delle annotations in Java 5, rende il processo di scrittura di un Test Case molto più semplice. Le annotazioni di Java 5 sono particolari istruzioni che consentono al programmatore di inserire metadati nel codice sorgente che rimangono accessibili anche in fase di esecuzione

mediante la tecnica della riflessione.

Le annotazioni di Junit ci permettono quindi di etichettare i metodi che vogliamo includere nei test case.

Le annotazioni che Junit fornisce sono le seguenti:

Annotazione	**Utilizzo**
@Test	È utilizzata per annotare i metodi che compongono la funzionalità da testare.
@Before	È utilizzata per annotare un metodo da eseguire prima dell'esecuzione di ogni test case (che contrassegnato con @Test), ad esempio per eseguire del codice prima dell'apertura di una connessione ad un database.
@After	È utilizzata per annotare un metodo da eseguire dopo l'esecuzione di ogni test case (che contrassegnato con @Test), ad esempio per eseguire del codice per la chiusura di risorse aperte in precedenza.
@BeforeClass	Stesso comportamento di @Before con la differenza che @BeforeClass viene chiamato solo una volta.
@AfterClass	Stesso comportamento di @After con la differenza che @ AfterClass viene chiamato solo una volta.
@Ignore	È utilizzata per ignorare un test saltando quindi la sua esecuzione. È utile in quanto ci consente di evitare di commentare il codice.

L'ordine di esecuzioni dei metodi che vogliamo testare che sono contrassegnati con l'annotazione `@Test` non è definito. Ciò che è rilevante è se il singolo metodo, alla fine dell'esecuzione, ha superato il test. In caso di esito negativo si dice che il test relativo a quel metodo è fallito.

In alcuni casi è necessario verificare che il metodo, in situazione di errore, sollevi una particolare eccezione oppure che esso non abbia superato un determinato tempo di esecuzione. A questi fini si utilizzano parametrizzazioni che si applicano alla clausola `@Test`.

Con `@Test (expected=Exception.class)` indichiamo a Junit di verificare che il metodo sollevi una eccezione e che essa sia come quella indicata (ad esempio `IOException`). Se il metodo non solleva una eccezione oppure ne solleva una diversa da quella indicata il test fallirà.
È utili nei casi in cui si vuole analizzare il comportamento del metodo in condizioni di errore.
Con `@Test(timeout=100)`, invece, indichiamo a Junit di verificare che l'esecuzione del metodo non ecceda una quantità di tempo espressa in millisecondi indicata nella parametrizzazione (ad esempio 100). Se il tempo impiegato per eseguire il metodo supera la soglia di tempo indicato il test fallirà.
È utile nei casi in cui il metodo sotto test esegua delle operazioni con l'esterno, ad esempio una interrogazione ad un database, una operazione real time o in generale in tutti quei contesti in cui la durata di esecuzione del metodo è un parametro critico da verificare.

Dopo aver introdotto i vari tipi di annotazioni che identificano i test passiamo ora alle istruzioni da inserire nel codice del metodo sotto analisi e che verificano determinate condizioni tali da imporre il successo o il fallimento del test.
Queste istruzioni sono metodi statici inclusi nella classe `org.junit.Assert` [7] che effettuano una comparazione tra il risultato elaborato dal metodo ed il risultato atteso: questo meccanismo è chiamato *oracolo* e rappresenta una delle fondamenta del testing del software. I metodi assert sono tutti metodi statici che non restituiscono alcun valore. Sono stati sviluppati diversi overloading per poter adattare Junit ai diversi contesti di utilizzo.
I metodi assert offerti sono i seguenti:

- assertEquals(atteso, attuale, [delta]): verifica che atteso e attuale siano uguali entro una differenza massima delta.
- assertNotEquals(atteso, attuale, [delta]): verifica che atteso e attuale non

siano uguali al di fuori di una differenza massimo delta.

- assertNull(attuale): verifica che attuale sia Null.
- assertNotNull(attuale): verifica che attuale non sia Null.
- assertSame(atteso, attuale): verifica che atteso e attuale facciano riferimento allo stesso oggetto.
- assertNotSame(atteso, attuale): verifica che atteso e attuale non facciano riferimento allo stesso oggetto.
- assertThat(attuale, Matcher): verifica che attuale soddisfi le condizioni specificate da Matcher. Matcher è una interfaccia fornita da Junit che consente di implementare specifici comparatori per oggetti più complessi.
- fail(): forza il fallimento del test

I parametri tra [] sono opzionali.
Tutti i metodi sopra citati accettano opzionalmente un ulteriore parametro di tipo stringa che consente di impostare il messaggio da visualizzare in caso il metodo chiamato abbia esito positivo.

Riprendendo l'esempio del metodo che converte un numero positivo in un numero negativo proposto in precedenza, una sua possibile implementazione è la seguente. È stato aggiunto il controllo sul parametro in input tale che se esso è uguale a zero viene sollevata una eccezione.

```
public int PosToNeg(int a) throws Exception {
        if(a == 0) throw new Exception();
        int b = a * (-1);
        return b;
}
```

Per creare un Test Case viene creata una classe di test contenente un metodo di test pubblico, void e senza parametri che analizza la correttezza di PosToNeg e

che viene chiamata PosToNeg_Test. Al nuovo metodo aggiungiamo la notazione @Test di Junit ed inseriamo i metodi assert con i quali verifichiamo la correttezza dell'implementazione:

```
@Test
public void PosToNeg_Test() throws Exception {
        int a = 5;
        int b = PosToNeg(a);
        assertTrue(b < 0);
}
```

È stata aggiunta l'istruzione assertTrue(boolean condition) che valuta la condizione booleana, se è negativa solleva l'eccezione di Junit determinando il fallimento del test. Nel nostro caso la condizione booleana restituisce true se b è minore di zero, quindi se il metodo funziona correttamente.
Per analizzare ulteriormente il comportamento del metodo è possibile aggiungere altri controlli: il primo controllo che aggiungiamo verifica che il numero in input sia positivo. Infatti se l'input è negativo, relativamente alla nostra implementazione, l'output sarà positivo e quindi errato.
Il codice è il seguente:

```
@Test
public void PosToNeg_Test() throws Exception {
        int a = -5;
        assertFalse("Input errato", a < 0);
        int b = PosToNeg(a);
        assertTrue(b < 0);
}
```

Abbiamo inserito anche un parametro opzionale che rappresenta la stringa che vogliamo visualizzare in caso la condizione valutata da assertFalse fallisca, ovvero se il numero in input è negativo. Il secondo controllo che aggiungiamo si basa sul fatto che il metodo preso in considerazione non può accettare un numero uguale a zero. In tal caso il metodo deve sollevare una eccezione che

deve corrispondere a quella definita nell'implementazione di `PosToNeg`.

```
@Test(expected=java.lang.Exception.class)
public void PosToNeg_Test() throws Exception {
        int a = 0;
        int b = PosToNeg(a);
        assertTrue(b < 0);
}
```

È stato quindi aggiunto il parametro expected alla clausola @Test con il nome dell'eccezione che ci aspettiamo che il metodo sollevi quando gli viene passato un numero uguale a zero. Naturalmente, in fase di esecuzione del test, alla chiamata del nostro metodo PosToNeg è necessario passare a uguale a zero. Qualora, passando zero, il metodo non sollevi alcuna eccezione o comunque una eccezione non del tipo specificato, il test fallirà indicando che il codice scritto non è corretto.

2.2 TestNG

Con l'aumento dell'adozione delle tecniche di testing sono aumentate anche le esigenze degli sviluppatori che hanno spinto la nascita di nuovi tool. Nel 2004 Cédric Beust realizza TestNG, un nuovo framework di testing per la programmazione in Java. TestNG è fortemente ispirato a JUnit e NUnit ma introduce delle nuove caratteristiche che tutt'ora lo rendono un forte concorrente di JUnit.

Prima di analizzare le differenze funzionali tra i due framework [9, 10] di testing esaminiamo le annotazioni che TestNG rende disponibili in aggiunta a quelle di JUnit.

Annotazione	**Utilizzo**
@BeforeSuite	È utilizzata per annotare un metodo da eseguire dopo che tutti i test della test suite sono stati eseguiti
@AfterSuite	È utilizzata per annotare un metodo da eseguire prima che tutti i test della test suite sono stati eseguiti
@BeforeTest	È utilizzata per annotare un metodo da eseguire dopo l'esecuzione di un altro test selezionato attraverso un tag
@AfterTest	È utilizzata per annotare un metodo da eseguire prima dell'esecuzione di un altro test selezionato attraverso un tag
@BeforeGroups	È utilizzata per annotare un metodo da eseguire dopo l'esecuzione di un gruppo di test selezionati attraverso i tag
@AfterGroups	È utilizzata per annotare un metodo da eseguire prima dell'esecuzione di un gruppo di test selezionati attraverso i tag

La seguente tabella mostra le principali differenze tra TestNG e JUnit. Da notare come attualmente JUnit sia arrivato alla versione 5 nella quale sono state introdotte nuove funzioni ma le differenze mostrate nel grafico continuano a persistere.

Tool	TestNG	JUnit 4
Annotation Support	Si	Si
Exception Test	Si	Si
Ignore Test	Si	Si
Timeout Test	Si	Si
Suite Test	Si	Si
Group Test	Si	No
Parameterized (primitive value)	Si	Si
Parameterized (object)	Si	No
Dependency Test	Si	No

La prima differenza di TestNG rispetto a JUnit è la possibilità di creare gruppi di metodi in modo semplice ed efficiente. Un gruppo è un insieme di test ed ogni test può appartenere ad uno o più gruppi. Questa caratteristica consente al programmatore di analizzare test che hanno delle relazioni intrecciate fra loro, testando i metodi appartenenti solo a determinati gruppi. Nella pratica i gruppi possono essere visti come un ulteriore livello tra i casi di test e la test suite. Un esempio di creazione di un gruppo in cui il test viene associato a due gruppi è il seguente:

```
@Test(groups = { "group1", "group2" })
```

Il concetto dei gruppi è stato ultimamente ripreso anche dagli sviluppatori di JUnit che hanno introdotto questa caratteristica sotto il nome di tag. Ogni metodo può essere taggato con uno o più etichette che vengono utilizzate per filtrare solo i metodi che possiedono quelle target. Una seconda caratteristica di

TestNG rispetto a JUnit riguarda la possibilità di eseguire diversi test in parallelo. È chiamata parallelismo ed è realizzata con il multithread. Ad esempio, per invocare 9 invocazioni di un pool di thread contenente 3 thread, l'annotazione è la seguente:

```
@Test(threadPoolSize = 3, invocationCount = 9)
```

In JUnit realizzare questa soluzione non è affatto rapida in quanto è necessario creare un metodo runner custom.

Un terzo aspetto in cui TestNG propone una soluzione diversa a JUnit riguarda il problema di fornire diversi valori di input allo stesso test. Il concetto alla base è la creazione di un array bidimensionale che raccoglie tutti i parametri che si vogliono passare, concetto condiviso anche nella soluzione di JUnit.

Ma in TestNG la definizione dell'array bidimensionale è implementata con l'annotazione `@DataProvider` con la quale viene identificato il metodo che genera l'array e lo restituisce in output. Successivamente basta richiamare il Data Provider attraverso il suo nome e passarlo come parametro all'annotazione `@Test` del metodo che vogliamo abbia in input i parametri generati. La definizione del Data Provider può anche essere fatta anche con un file XML che include al suo interno tutti i parametri da sottoporre in input al metodo target. Questa funzione rende TestNG fortemente integrabile nel processo di testing di un prodotto software perché molto spesso i casi di test sono prodotti da tool esterni e vengono forniti in un file XML.

Quarto aspetto di differenza di TestNG rispetto a JUnit è rappresentato dalle dipendenze tra gruppi e/o metodi. La dipendenza fornisce un metodo per analizzare se le varie chiamate dei metodi rispettano un ordine previsto. TestNG consente di dichiarare dipendenze tra test ed include la possibilità di saltare il test se la dipendenza non è verificata.

Questa funzionalità non è inclusa in JUnit ma può essere emulata usando le as-

sunzioni. Quando una assunzione fallisce, fallisce anche il test che viene quindi ignorato.

Il successo che TestNG sta riscontrando in questi ultimi anni è dovuto al fatto che in alcuni aspetti è oggettivamente superiore. Questo è il motivo per cui è stato scelto per questo progetto. Infatti il codice implementato che si descriverà in seguito, produrrà un file avente dell'altro codice pensato per essere eseguito con il plugin TestNG per Eclipse.

2.3 Combinatorial Testing

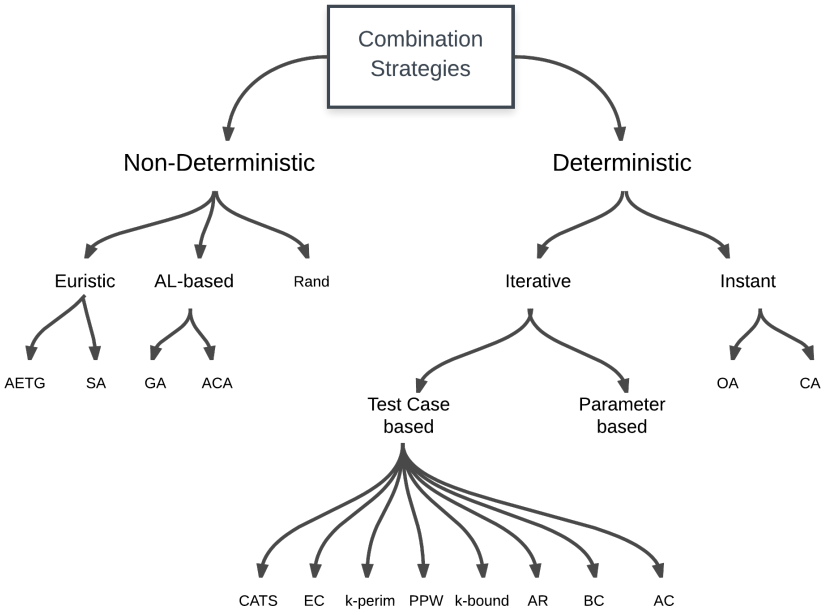
Il Combinational Testing si pone come nuova tecnica anteponendosi alle tecniche classiche di scrittura dei test fortemente legate al contesto applicativo.

Si supponga di avere un sistema, non necessariamente di tipo software.

In uno scenario realistico, in cui il fattore di utilizzo del prodotto aumenta significativamente, può accadere di individuare problemi che fino a quel momento erano rimasti nascosti. Banalmente, un sistema con molti parametri, mostra errori o difetti dovuti alla loro combinazione che prima non era stata verificata. Una soluzione a questo problema potrebbe essere l'analisi esaustiva di tutte le combinazioni. Questa opzione però non rappresenta la soluzione al problema in quanto spesso è inutile e molto dispendiosa in termini di tempo e di costo.

Il concetto alla base del Testing Combinatoriale è che non tutti i parametri dell'insieme totale delle combinazioni contribuiscono all'errore perché molti di loro sono causati dall'interazione di un ristretto numero di parametri. Quindi il Testing Combinatoriale si occupa di determinare delle tecniche al fine di rendere questo restringimento del dominio efficiente ed efficace in fase di testing.

Esistono vari strategie combinatorie caratterizzate dal tipo di test che si vogliono verificare e dall'algoritmo utilizzato per individuare questi test.



Le strategie non deterministiche hanno una componente casuale per la determinazione dei test ed una loro proprietà è quella di produrre differenti Test Suite ad ogni esecuzione.

La tecnica più semplice prevede una selezione casuale.

Le strategie deterministiche, invece, producono gli stessi test ad ogni esecuzione.

È possibile costituire una terza strategia come la combinazione delle precedenti, quindi con componenti deterministiche e non deterministiche.

Una proprietà del Testing Combinatoriale è il numero di elementi correlati fra loro che è necessario fornire come input per identificare un errore (o almeno per massimizzare la probabilità di trovarne). Si indica con t ed indica il grado di accoppiamento dei parametri del sistema sotto verifica.

Generalmente assume valore 2 ed in questo caso si dice pairwise testing, altrimenti se t è maggiore di 2 si dice t-wise testing.

Illustriamo il combinatorial testing attraverso la descrizione di un suo specifico

caso, il pairwise testing, che tuttavia rappresenta il maggior uso di questa tecnica.

Supponiamo di avere un software in cui le variabili di funzionamento variano tra Windows o Linux come sistema operativo, Intel o AMD come processore e IPv4 o IPv6 come protocollo internet. Con un rapido calcolo si ottengono $2x2x2 = 8$ casi possibili che è necessario testare sul software al fine di scovare errori.

Con la tecnica del testing pairwise, in cui il numero di elementi fra loro correlati e che si vuole analizzare è due (2-wise), il numero di test da eseguire passa da 8 a 4, come mostrato nella seguente tabella.

Configurazione Test Pairwise			
Test Case	Sistema Operativo	Processore	Protocollo Internet
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	Intel	IPv6
4	Linux	AMD	IPv4

In altre parole con il testing pairwise copriamo tutti le possibili coppie (elementi accoppiati t=2) di valori tra i tre parametri (Sistema Operativo, Processore e Protocollo Internet). Da notare che le combinazioni che soddisfano il testing t-wise con t=3 non sono state testate, come ad esempio Windows; Intel; IPv6 e sono composte dalle 4 combinazioni in tabella più le restanti 4 combinazioni non incluse.

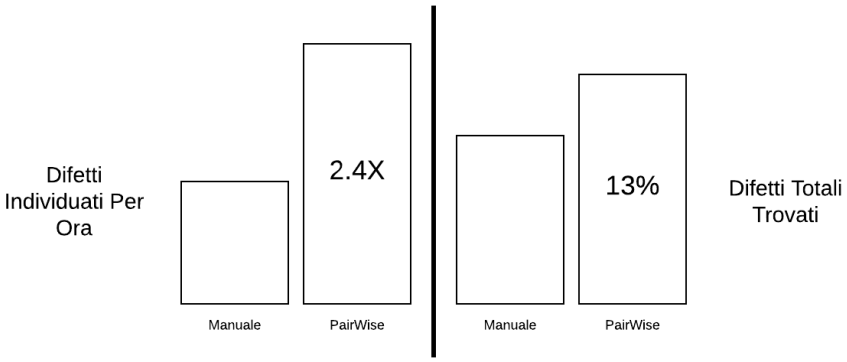
Sebbene in questo semplice caso la loro eliminazione non comporti vantaggi significativi, in casi più complessi il vantaggio è realmente percepibile. Ad esempio, in un sistema di automazione industriale in cui si hanno 20 controlli ognuno

con 10 impostazioni diverse, si avrebbero 1020 combinazioni, che per essere verificate necessitano di più tempo della vita stessa dello sviluppatore addetto al testing.

Sorprendentemente, un controllo con tecnica t-wise con t=3, prevede solamente 180 test, se accuratamente costruiti.

È stato condotto uno studio con l'obiettivo di evidenziare le differenze tra l'uso di metodi classici di scrittura di test e l'uso del Combinational Testing di tipo pairwise. Con metodi classici si intendono le tecniche con cui lo sviluppatore scrive casi di test fortemente legati alle specifiche ed alle funzionalità del programma.

I risultati sono netti ed evidenziano come i metodi di test combinatoriale sono quasi tre volte più efficienti dei metodi classici. Come mostrato nei grafici di seguito, individuano 2,4 volte più difetti per ora e raggiungono una maggiore copertura. Infatti al completamento dei test, i metodi combinatoriali individuano il 13% in più di difetti rispetto ai metodi classici.

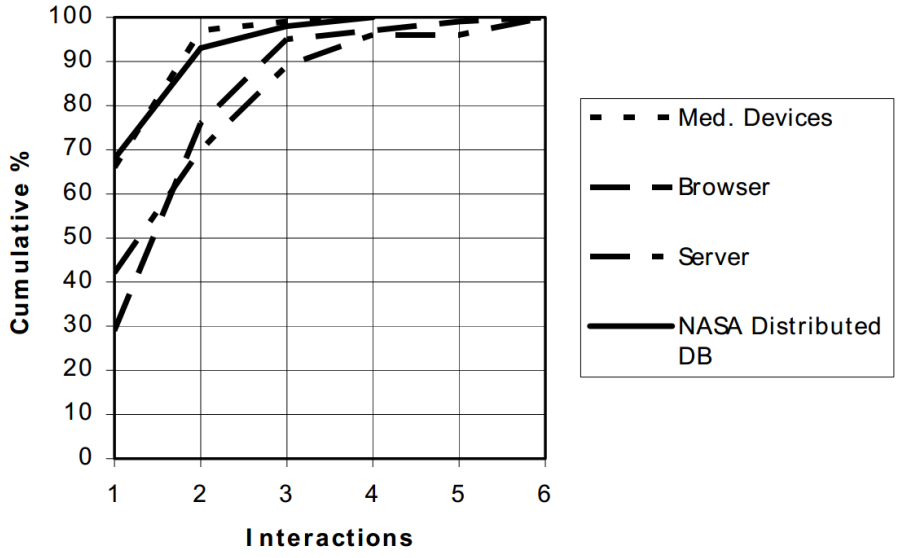


Ma il test combinatoriale si basa sul test di un restringimento del dominio di input del sistema, cosa dire della restante parte del dominio?

Innanzitutto va specificato che il grado di accoppiamento da scegliere in fase di testing è fortemente legato al contesto.

In uno studio del 1999 [4], che analizza i guasti in condizioni rare su apparecchiature mediche, il National Institute of Standards and Technology (NIST) ha individuato un caso in cui l'errore si determinava con la combinazione di ben 4 parametri. Un test pairwise non avrebbe mai individuato quell'errore perché lo avrebbe escluso a priori dal dominio dei casi da effettuare imponendo un grado di accoppiamento uguale a 2, e non 4.

Investigando su altri tipi di applicazioni con distribuzioni simili di errori si è notato come la maggior parte dei difetti fosse causata da un singolo parametro, una minore parte dall'interazione dei valori di due parametri e, progressivamente, una sempre minore parte dall'interazione dei valori di 3, 4, 5 e 6 parametri. Il grafico di seguito mostra la diversa distribuzione della percentuale di errori trovati in funzione del grado di accoppiamento in quattro diversi ambiti applicativi: dispositivi medici, browser, web server e database distribuito della NASA [2].



Come anticipato inizialmente, il grado di accoppiamento uguale a 2 soddisfa, nella maggior parte dei casi, fino al 98% degli errori presenti nel sistema sotto

osservazione. Nel caso del database distribuito della NASA e dei dispositivi medici, con una interazione uguale a 4 si raggiunge praticamente il 100% di errori trovati. Nel caso del web server, invece, si nota come un aumento del grado di accoppiamento da 4 a 5 non comporta alcun miglioramento in termini di errori individuati, incidendo invece in modo significativo sul tempo e sulla elaborazione dei test selezionati.

L'ingrediente chiave per questo tipo di testing è un array di copertura, ovvero un oggetto matematico che rappresenta la copertura di grado t di tutte le combinazioni dei parametri, combinate almeno una volta.

Per l'esempio mostrato in precedenza con `t=2`, la generazione del relativo array di copertura è di semplice elaborazione. Invece, per array di copertura con grado di interazione maggiore fino al 6, la loro elaborazione è molto più difficile al punto che solo con i nuovi algoritmi è possibile determinarla.

Un esempio di array di copertura è il seguente in cui è stato impostato un grado di accoppiamento `t=3` per 10 parametri di tipo binario, ovvero che ogni parametro può assumere solo due valori, nel nostro caso 0 o 1. I casi di test risultano essere solo 13.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	1	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

In questo caso un testing 3-wise individua approssimativamente il 90% di errori in tutti e 4 gli studi empirici inerenti al grafico degli andamenti sopra presentato. Per una copertura esaustiva degli errori, ovvero tutte le combinazioni possibili e cioè con dominio di test uguale al dominio totale, sarebbero necessari 210 test, ovvero 1024 test. Facendo una analisi su questo esempio che presenta 10 parametri ognuno con due soli possibili valori ed ipotizzando che ci siano in totale 10 difetti, abbiamo che con 13 casi di test (3-way coverage), individuiamo 9 errori e con 1024 casi di test (100% coverage), individuiamo 10 errori.
Analizzando il loro rapporto,

$$\frac{9}{13} : \frac{10}{1024}$$

mostriamo come con 3-way coverage troviamo 71 volte più difetti per caso di test che con una copertura totale, aumentando di netto l'efficienza del testing.

3 Un framework per Cit

3.1 CitLab

CitLab è un tool per testing combinatoriale che consente di importare ed esportare modelli di problemi combinatoriali da e per differenti domini applicativi. È stato concepito e sviluppato da Paolo Vavassori, dottorando dell'Università degli Studi di Bergamo e da Angelo Michele Gargantini docente di Testing e Verifica del Software nella medesima università.
Il suo principale obiettivo è quello di fornire un linguaggio astratto comune a tutti coloro che intendono affrontare il problema del combinatorial testing con un ambiente di sviluppo fortemente estendibile.
Al contrario di altri tool sviluppati per il testing combinatoriale [3], CitLab offre la possibilità di costruire il modello attraverso un linguaggio definito e non attraverso una interfaccia grafica.
È inoltre progettato per essere fortemente integrato con Eclipse IDE attraverso il meccanismo delle estensioni con plug-in. CitLab consente di usare, sulla stessa test suite, diversi algoritmi per generare casi di test che poi possono essere confrontati fra loro.
Le funzionalità di CitLab:

- È un linguaggio astratto, ricco e con precise regole semantiche specificatamente costruito per problemi combinatoriali.
- Ha una sintassi concreta ed una grammatica ben definita per consentire la scrittura di modelli e la loro condivisione attraverso una notazione comune.
- È un framework basato su Eclipse Modeling Framework (EMF) che fornisce gli strumenti ed il supporto runtime per la generazione automatica di un set di classi in Java per il modello combinatoriale unitamente ad

un set di classi di supporto e di utilità che consentono di interagire con il modello attraverso API.

- Possiede un editor integrato in Eclipse IDE per la creazione e la modifica di problemi combinatoriali. Ha con le principali caratteristiche di un moderno editor di programmazione, quali l'evidenziazione della sintassi, il completamento del codice, il controllo run-time degli errori, la correzione rapida e la vista outline del codice.

- Ha una ricca collezione di metodi e classi di utilità, specificatamente sviluppate per i problemi combinatoriali in CitLab, per la manipolazione di modelli e per la generazione di test suite.
  Ad esempio CitLab fornisce un metodo per la generazione di tutti i test che forniscono una copertura combinatoriale di grado t.

- È un framework che consente l'aggiunta di nuovi algoritmi per la generazione di casi di test attraverso plugin. Questo permette ai ricercatori di sviluppare ed analizzare nuovi tecniche all'interno del contesto CitLab senza dover definire una propria grammatica, un parse o risolvere altri problemi non strettamente legati al Testing Combinatoriale.

- È un framework che, attraverso la traduzione di codice, permette di importare ed esportare modelli e casi di test espressi in altre notazioni sfruttando le trasformazioni Model to Test (M2T) o Model to Model (M2M).

La struttura di un modello CitLab consiste in sei parti: *Definitions*, *Types*, *Parameters*, *Constraints*, *Seeds*, *TestGoals* e *ExpectedResult* .

Definitions: in questa sezione l'utente può definire costanti numeriche che possono essere usate come nella definizione di vincoli, test goal e seeds.

Es: `Number threshold = 27;`

Parameters: in questa sezione possono essere definiti i parametri (input) del

sistema. I tipi di parametri possono essere di tipo enumerativo stringa, booleano, range, enumerativo intero.

Types: è possibile definire un nuovo tipo dichiarandolo in questa sezione. Dopo la definizione del nuovo tipo esso può essere usato in altre sezioni del modello.

Constraints: consentono di includere dei vincoli da imporre al modello. Se il modello non li rispetta esso viene considerato non valido.

È un utile strumento per restringere il dominio della test suite ai soli test che verifica positivamente le Constraints.

Per esprimere le Constraints è utilizzato un linguaggio di logica proposizionale con simboli di aritmetica. L'inserimento di più vincoli implica che essi devono essere verificati tutti contemporaneamente.

Seeds: rappresentano i test che l'utente vuole eseguire in ogni caso o test già eseguiti in precedenza che impongono un ulteriore vincolo sul modello ed allo stesso tempo restringo il dominio di appartenenza dei test.

ExpectedResult : in questa sezione sono incluse le istruzioni che rappresentano i valori attesi del modello.

La sintassi per definire un Expected Result è la seguente:

`<nomecontrollo> checks ( <operazioni aritmetiche> )`

ed un esempio è:

Es: `<somma> checks ( (a + b) )` con `a` e `b` parametri.

All'interno delle parentesi è possibile definire delle operazioni matematiche che coinvolgono i parametri definiti nella sezione `Parameters`. CitLab valuta queste operazioni e calcola i valori attesi per ciascuna combinazione di parametri generata, ovvero da ciascun test della Test Suite.

Questa funzionalità di CitLab permette di analizzare la corretta implementazione del software modellizzato sfruttando il plugin TNG Exporter, oggetto di questa tesi e illustrutato in dettaglio nei seguenti capitoli.

Gli algoritmi che CITLAB usa per generare i casi di test non sono direttamente inclusi ma sono integrati attraverso la tecnica dei plugin. In questo modo è possibile aggiungere nuovi algoritmi semplicemente aggiungendo il relativo plugin.
Attualmente, i plugin supportati sono i seguenti:

- AETG: è un plugin che segue lo pseudo codice implementato per algoritmi greedy.
- IPOs: è un plugin sviluppato dagli stessi sviluppatori di CITLAB [14]. È un nuovo algoritmo euristico parametrizzato per la costruzione di test suite con copertura pairwise (`t=2`). Sfrutta la proprietà di simmetria degli array di copertura sopra descritti.
- Random: è un semplice algoritmo che aggiunge nuovi casi di test random fino a che non viene raggiunta la copertura.
- ACTS: è un tool esterno per la generazione di test sviluppato dal NIST.
- CASA: è un tool esterno per la generazione di test sviluppato da Myra Cohen e suoi colleghi e basata sulla Simulated Annealing (ricottura simulata) [6].
- ATGT_SMT: è un tool esterno che combina euristica e SMT (satisfiability modulo theories).

CitLab fornisce tre plugin che consentono l'esportazione dei casi di test generati, ognuno con un proprio formato. Tutti e tre i formati condividono la logica di esecuzione che prevede la scrittura del modello, la generazione dei casi di test con uno specifico algoritmo e la scelta del formato e del nome dell'oggetto da esportare.

Il primo formato messo a disposizione da CitLab è il comune `csv`, comma separated values, che dispone un caso di test per riga. Ogni caso di test presenta la combinazione di input relativa al caso di test separando con il carattere virgola ciascun elemento. È un formato molto comune per il quale esistono molti software che consentono una rapida importazione dei dati formattati csv.
Il secondo formato messo a disposizione è `xls`, ovvero il formato dei fogli di calcolo di Excel. Consente una rapida visualizzazione dei casi di test generati e la possibilità di elaborarli con gli strumenti propri di Excel. Non è necessaria alcuna ulteriore operazione sul file al fine di risolvere problemi di compatibilità: just click and see.
Il terzo formato reso disponibile da CitLab è utile a coloro che vogliono eseguire i test generati sul software da loro precedentemente modellizzato. Richiede il modello del sistema sotto analisi come input e genera le classi da includere nel proprio progetto per applicare i casi di test al sistema. Questo aspetto è stato implementato nel progetto di cui questo documento vuole essere una descrizione del contesto del problema, del problema nello specifico e della soluzione sviluppata a livello di codice.

3.2 Extension Point

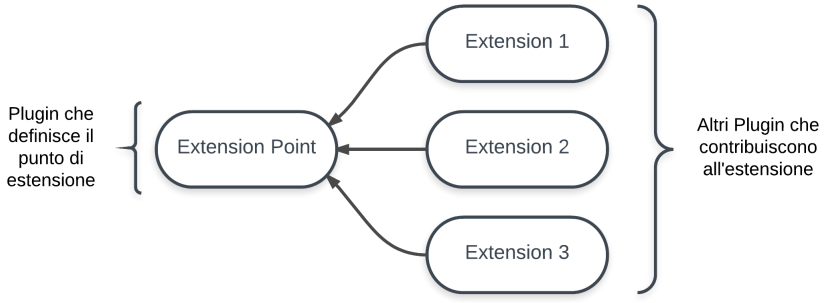
Una regola di base per la costruzione di sistemi software modulari è quello di evitare un forte accoppiamento tra i componenti stessi. Se i componenti sono strettamente integrati diventa difficile rassembrarli in configurazioni differenti o di sostituire un componente con un altro avente una diversa implementazione senza causare un problema che può, a cascata, ripercuotersi su tutto il sistema. In Eclipse, per sviluppare un accoppiamento di componenti in modo non forte, si utilizza il meccanismo delle estensioni ed i punti di estensione.
Se volessimo fare un esempio di ciò che i punti di estensione possono essere nella realtà, li potremmo paragonare alle prese elettriche. La presa è l'extension

point e la spina della lampadina a cui si collega è l'estensione. Come con le prese elettriche, i punti di estensione sono disponibili in diverse forme e dimensioni, e solo le estensioni che sono progettate per quel particolare extension point si adattano correttamente.

Tradotto nel nostro contesto, questo significa che quando un plug-in vuole consentire ad altri plug-in di estendere o personalizzare delle sue funzionalità, come nel caso in oggetto con l'implementazione del plugin TNG Exporter, esso dichiarerà un punto di estensione al quale gli altri plug-in, le estensioni, si agganceranno.

Il punto di estensione dichiara formalmente un contratto composto in genere da una combinazione di XML e interfacce Java, alle quali le estensioni devono essere conformi. I plug-in che vogliono connettersi a questo extension point devono implementare a loro volta, nella loro estensione, il contratto.

Un concetto fondamentale è che il plug-in che viene esteso non sa nulla riguardo i plug-in che lo estendono, se non che essi devono rispettare il suo contratto che definisce nel punto di estensione. Questo permette a plug-in, costruiti da diversi attori, di interagire e funzionare correttamente senza che essi abbiano uno stretto accoppiamento.



La piattaforma Eclipse è fortemente basata sul concetto di estensione e punto di estensione. Alcune sue caratteristiche sviluppate come plugin sono completamente dichiarative, ovvero non forniscono funzionalità sviluppate con del codice.

Ad esempio, Eclipse possiede un punto di estensione che fornisce la funzionalità per la personalizzazione della combinazione di tasti, un altro definisce le annotazioni personalizzate nei file, chiamati marcatori.
Un'altra tipologia di extension point consente di sovrascrivere il comportamento predefinito di un determinato componente. Ad esempio, gli strumenti di sviluppo di codice Java includono un formattatore di codice che espone un punto di estensione attraverso il quale plugin di terze parti si collegano e forniscono la funzionalità in sua sostituzione.
Una ulteriore categoria di punti di estensione è utilizzata per gestire finestre ed altri elementi relativi all'interfaccia utente. Per esempio i punti di estensione per l'interfaccia grafica che forniscono viste, editor e procedure guidate [8].

4 TNG Exporter

4.1 Obiettivo

Il progetto di TNG Exporter consiste nel realizzare un plugin per esportare la Test Suite, ottenuta con tecniche combinatoriali in CitLab, in una classe Java che rappresenta un caso di test di TestNG. La classe prodotta deve essere completamente eseguibile nel framework di TestNG.
Lo sviluppatore che utilizza TNG Exporter ha così la possibilità di verificare la corretta implementazione del suo codice relativo al modello CitLab, confrontando i valori attesi ottenuti dal modello con i valori restituiti dal suo codice.
Il test è realizzato con l'ausilio di TestNG, le sue annotazioni `@DataProvider` e `@Test`, e l'inserimento dei suoi metodi Assert nel codice generato dal plugin.

4.2 Descrizione Logica di Funzionamento

Lo scenario di utilizzo di TNG Exporter ipotizza che lo sviluppatore, utilizzatore finale, abbia modellizzato il proprio sistema software (o il suo metodo target) in CitLab. CitLab consente, come descritto nei paragrafi precedenti, di generare una Test Suite a partire da un suo modello, impostando algoritmo di generazione e grado di accoppiamento dei parametri.
È possibile genere i valori attesi formulando espressioni matematiche da inserire nella sezione `ExpectedValue` del modello CitLab.
TNG Exporter ottiene le combinazioni generate ed i relativi valori attesi dalla Test Suite, ottenuti dal plugin CitLab secondo la logica di associazione dell' Extension Point descritta nei capitoli precedenti.
Affinchè lo sviluppatore abbia a disposizione una classe in Java pronta all'uso, si è deciso di utilizzare il tool TestNG a supporto delle operazioni di test, e di sviluppare l'associazione Data Provider-Test per fornire le combinazioni generate ed i valori attesi della Test Suite al metodo che contiene i confronti dei valori

ottenuti con i valori attesi.

La classe finale comprende quindi un metodo che fornisce il Data Provider e un metodo in cui lo sviluppatore aggiunge i test desiderati avendo a disposizione il singolo test della Test Suite formato da combinazione dei parametri e valori attesi.

Alcune opzioni relative a parametri di generazione della classe possono essere impostati dallo sviluppatore modificando il file `options.json` che il plugin leggerà in fase di esecuzione.

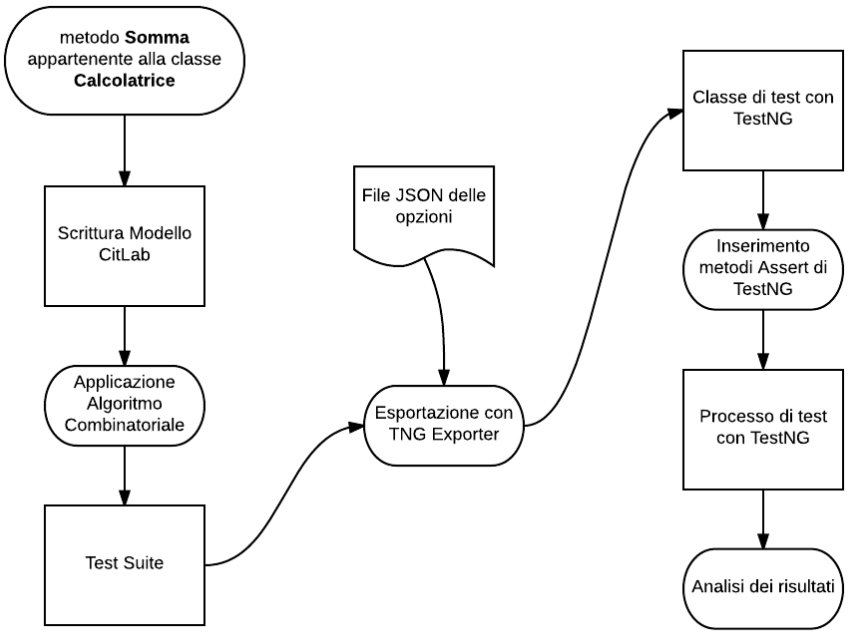


Figura 1: Flusso del processo di testing: dal modello CitLab alla classe di test combinatoriale.

4.3 Caso di Uso

Si suppone di avere l'implementazione di una classe avente al suo interno un metodo che implementa una funzionalità.

Si decide di modellizzare il metodo con CitLab, dichiarando i parametri di ingresso nella sezione `Parameters` ed il valore atteso nella sezione `ExpectedResults`.
Ad esempio una classe `Calcolatrice` contiene un metodo di nome `Somma` che implementa la fuzionalità della somma. Questo metodo richiede in input due numeri interi che per motivi di praticità consideriamo compresi tra 1 e 5. Come risultato `Somma` restituirà la somma $a+b$ che corrisponde al nostro valore atteso.
Il modello CitLab sarà:

```
Parameters:
        Range a [ 1 .. 5 ];
        Range b [ 1 .. 5 ];
end
ExpectedResults:
        somma checks ( (a+b) );
end
```

Quindi, con l'algoritmo scelto, si genera la Test Suite che conterrà come casi di test tutte le possibili combinazioni dei due parametri `a` e `b`.
Nell'esempio qui proposto si otterranno 25 combinazioni totali.
Per procedere al test del metodo `Somma` della classe `Calcolatrice` si decide di utilizzare il plugin TNG Exporter al fine di ottenere una classe `Java` subito eseguibile.
È però necessario fornire a TNG Exporter il percorso del file `options.json` nel quale abbiamo definito il `package`, il nome della classe e del metodo di test da generare e il nome del Data Provider da utilizzare.
È infatti possibile sfruttare queste opzioni per includere la classe da generare in un package già esistente, ad esempio dedicato al test, che è buona norma.
Oppure selezionare il nome di un Data Provider da sfruttare in altri metodi di test all'interno dello stesso progetto.
Ottenuto il percorso del file delle opzioni, TNG Exporter creerà all'interno del progetto e del `package` selezionato, un file in formato `Java` contenente il Data Provider ed il metodo di test.

In questo caso sarà:

```
package packagename;
import org.testng.annotations.*;
class ClasseSommaTest {
        @DataProvider(name = "
            ProviderCombinazioniInteriAB")
        public Object[][] rangeData() {
                /*a, b || somma*/
                return new Object [] [] {
                        { 1, 1, 2.0 },
                        { 1, 2, 3.0 },
                        { 1, 3, 4.0 },
                        { 1, 4, 5.0 },
                        { 1, 5, 6.0 },
                        { 2, 1, 3.0 },
                        { 2, 2, 4.0 },
                        { 2, 3, 5.0 },
                        { 2, 4, 6.0 },
                        { 2, 5, 7.0 },
                        { 3, 1, 4.0 },
                        { 3, 2, 5.0 },
                        { 3, 3, 6.0 },
                        { 3, 4, 7.0 },
                        { 3, 5, 8.0 },
                        { 4, 1, 5.0 },
                        { 4, 2, 6.0 },
                        { 4, 3, 7.0 },
                        { 4, 4, 8.0 },
                        { 4, 5, 9.0 },
                        { 5, 1, 6.0 },
                        { 5, 2, 7.0 },
                        { 5, 3, 8.0 },
                        { 5, 4, 9.0 },
                        { 5, 5, 10.0 }
                };
        }
        @Test(dataProvider = "
            ProviderCombinazioniInteriAB")
        public void TestSomma(Object a, Object b, Object
            somma) {
                System.out.println("a: " + a + ", b: " +
                    b + " --> somma: " + somma);
                /* Add your asserts here */
                }
        }
```

Il metodo `rangeData` restituisce, come matrice di `Object`, tutte le combinazioni possibili degli input con affiancati sulla destra il corrispondente valore atteso. Ad esempio, con $a = 3$ e $b = 5$ il valore atteso risulta essere $somma = 8$. All'interno del metodo `TestSomma` verranno inseriti i metodi `Assert` di TestNG che, opportunamente costruiti, consentiranno di verificare l'oracolo, ovvero la positività del confronto tra valore ottenuto e valore atteso.

Un esempio è il seguente

```
Assert.assertEquals((int)a+(int)b, (int)(double)somma, "
    Metodo Somma implementato correttamente");
```

Successivamente si esegue la classe `ClasseSommaTest` con TestNG che fornirà i risultati dei test effettuati, ovvero se il metodo della somma è stato implementato correttamente oppure no.

4.4 Implementazione

Il primo passo per implementare TNG Exporter è stato quello di creare il plugin seguendo le specifiche descritte nel capitolo dell'Extension Pointer. È stata aggiunta l'estensione all'Extension Pointer esposto da CitLab ed implementata una classe per rispettare il contratto dichiarato dall'Extension Pointer.

La principale classe implementata è chiamata `TngTestSuiteExporter` ed estende `ICitLabTestSuiteExporter` che è contenuta in `citlab.core.ext`. `ICitLabTestSuiteExporter` contiene la definizione di un metodo astratto `void` con la signatura `generateOutput(TestSuite input, String FileName)` i cui parametri in input sono la TestSuite generata da CitLab e il nome del file del modello da esportare che ha formato, nel nostro caso, di un file di testo con estensione `.java`.

La signatura completa è:

```
public abstract void generateOutput(TestSuite input,
    String FileName);
```

`TngTestSuiteExporter` esegue l'override del metodo `generateOutput` che, con i parametri forniti in input sopra descritti, costruisce la classe java contenente il metodo che elabora il test dei parametri ed il DataProvider che fornisce l'array bidimensionale.

La prima istruzione scritta nel metodo `generateOutput()` è un blocco try-catch per la cattura di eventuali errori. I tipi possibili di eccezioni che possono presentarsi sono unicamente di tipo Input/Output perché è necessario creare un file di testo e le operazioni relative a queste istruzioni possono essere bloccate per motivi esterni, ad esempio memoria insufficiente o nome del file non valido. Tutte le istruzioni di seguito descritte sono inserite all'interno del blocco try-catch.

Le prime sono inerenti alla creazione del file java ed all'inizializzazione del gestore per la scrittura nel file creato. Successivamente si è inizializzato un oggetto di tipo `ExpectedValueGetter`, implementato nelle classi di utilità di CitLab, per l'ottenimento dei valori attesi di una Test Suite. Questi valori, che sono gestiti internamente come una lista di stringhe, sono ottenuti dalle istruzioni inserite nel campo `ExpectedResults` del modello CitLab.

Alcuni parametri della classe devono essere impostati dallo sviluppatore, come ad esempio il nome della classe ed il package in cui è contenuto. Si è quindi deciso di consentire di impostare queste opzioni attraverso la creazione di un file JSON inserito nel progetto. Lo sviluppatore, dopo aver modificato il file JSON, fornisce il suo percorso attraverso una finestra di dialogo. Quindi il codice contenuto in `TngTestSuiteExporter` legge il file JSON e adatta la classe alle esigenze dello sviluppatore.

Come è stato anticipato, il formato di questo file è JSON che è l'acronimo di

JavaScript Object Notation. È un semplice formato per lo scambio di dati che si basa su un sottoinsieme del linguaggio di programmazione JavaScript (ECMA-262). Sono stati sviluppati molti gestori di questo formato per quasi la totalità dei linguaggi di programmazione [11].
Si basa su due tipi di strutture: la prima su un insieme di coppie/valore e la seconda su un elenco ordinato di valori.
In questo contesto i dati da gestire sono pochi e semplici ma si è comunque deciso di utilizzare questo formato per la semplicità nella sua gestione, dovuta all'esistenza di librerie Java consolidate, e per la possibilità, in futuro, di un maggior arricchimento dei dati.
Di seguito il contenuto del file JSON con valori di esempio:

```
{
        "Version" : "0.1",
        "Package": "packagename",
        "ClassName": "ClasseSomma",
        "DataProviderName": "NomeProvider",
        "TestName": "TestNumero1"
}
```

Le istruzioni per ottenere i parametri scritti nel file JSON sono le seguenti: come prima cosa si è inizializzato un oggetto per la gestione dei file in lettura impostando il percorso dove risiede il file di opzioni. Successivamente se ne controllo l'esistenza, che nel caso in cui non sia verificata, mostra a video una finestra di avviso. Se invece il file di opzioni esiste esso viene letto e, per la sua interpretazione, viene utilizzata una libreria di nome Gson scritta da Google.
Gson è una libreria per la serializzazione e deserializzazione per la conversione di un oggetto Java in JSON e viceversa. Mette a disposizione due metodi di nome `toJson()` e `fromJson()` per eseguire queste operazioni [12].
Nel nostro caso il metodo necessario è `fromJson()` che richiede in input il testo JSON da deserializzare e la classe che rappresenta la struttura del file JSON. È stata quindi creata una classe di nome `TngTestSuiteExporterOptionsFile`

con i campi `Version`, `Package`, `ClassName`, `DataProviderName` e `TestName`, i relativi metodi `set` e `get`, ed il costruttore per inizializzarli.

Il passo successivo riguarda l'insieme di istruzioni per la creazione della matrice bidimensionale contenente la matrice dei valori attesi accostata a destra alla matrice dei parametri della Test Suite, parametri tra loro combinati secondo le impostazioni fornite a CitLab.

I parametri tra loro combinati sono contenuti nella Test Suite fornita in input al metodo in oggetto e sono rappresentati con una lista di liste di oggetti di tipo `Assignment`. Un insieme di Assignment compone un Test, ed un insieme di Test compone la Test Suite.

I valori attesi possiedono una costruzione simile. Dopo essere stati ottenuti dalla classe di utilità inclusa in CitLab che si chiama `ExpectedValueGetter`, i valori sono strutturalmente composti da lista di liste di stringhe.

Prima di proseguire viene effettuato un controllo sulla lista di Test e sulla lista dei valori attesi. Esse devono avere uguale lunghezza ed inoltre la lista dei valori attesi non deve essere vuota. Questa verifica viene effettuata mediante il costrutto `assert` di Java.

Questo costrutto è stato introdotto dalla versione 1.4 del Java Development Kit e fornisce uno strumento efficace che lo sviluppatore può usare per controllare che in determinati punti del codice delle condizioni rimangano verificate.

Se il controllo restituisce esito negativo il programma termina.

Nel nostro caso l'istruzione è:

```
assert (ts.getTests().size() == expG.getExpectedValues().
    size() || expG.getExpectedValues().isEmpty());
```

A questo punto si hanno a disposizione tutti gli oggetti necessari per la creazione del file `.java` che avrà la seguente composizione

- Package ed elenco degli import

- Intestazione della classe di test, contenente:
  - Metodo `rangeData()`, annotato come DataProvider che restituisce una matrice bidimensionale.
  - Metodo con nome definito dallo sviluppatore, che effettua il test con i dati ottenuti dal Data Provider ed è annotato come `@Test`.

La definizione del `package` è scritta in funzione del valore definito dallo sviluppatore al momento della compilazione del file `options.json`.
Invece, gli unici `import` inclusi, sono inerenti a TestNG e consentono a TestNG stesso di riconoscere il metodo da testare ed il suo Data Provider.
Successivamente è stato scritto il codice per la generazione dell'intestazione della classe e del metodo `rangeData` che non ha valori in ingresso e restituisce un vettore di vettori di tipo `Object`, l'oggetto padre di tutti gli altri oggetti in Java. Questo metodo è contrassegnato con l'annotazione `@DataProvider` che ha come parametro il suo nome ottenuto dal file `options.json`. È attraverso il nome che è possibile legarlo al metodo che effettivamente esegue il test sul Data Provider, ovvero quello contenente le chiamate ai metodi statici `Assert` di TestNG.
Prima del codice per la costruzione del Data Provider sono state inserite, per motivi di leggibilità, le istruzioni per creare un commento di descrizione con i nomi delle colonne della matrice bidimensionale, ovvero i nomi dei parametri dei valori combinati e i nomi dei valori attesi, entrambi ottenuti dal modello CitLab.
Quindi è stato scritto il blocco di codice per generare la matrice bidimensionale. Iterando tutti i Test della Test Suite, vengono stampati nel file di destinazione tutti gli Assignment del test, ovvero tutti i parametri che compongono quel determinato Test e, di seguito, tutti valori attesi relativi a quel Test.
La matrice bidimensionale restituita ha dimensione $M \times N$ con M uguale al numero di test della Test Suite, che corrispondono al numero totale di com-

binazioni ottenute in CitLab, e N la somma del numero di parametri tra loro accoppiati che sono stati impostati in CitLab ed i valori attesi dichiarati, sempre nel modello CitLab.

È necessario formattare correttamente questi dati secondo la sintassi imposta da Java. Ricordiamo come il codice di cui si sta descrivendo la logica ha l'obiettivo di generare un file contenente dell'altro codice che deve essere completamente compilabile.

Il codice prodotto nel file .java è simile al seguente, naturalmente con i valori della matrice diversi per ogni modello CitLab.

```
@DataProvider(name = "NomeProvider")
public Object[][] rangeData() {
        /*primo, secondo || sum, double_sum*/
        return new Object [] [] {
                { 12, 25, 37.0, 74.0  },
                { 37, 25, 62.0, 124.0 },
                { 44, 25, 69.0, 138.0 },
                { 12, 26, 38.0, 76.0  },
                { 37, 26, 63.0, 126.0 },
                { 44, 26, 70.0, 140.0 },
                { 12, 27, 39.0, 78.0  },
                { 37, 27, 64.0, 128.0 },
                { 44, 27, 71.0, 142.0 },
                { 12, 28, 40.0, 80.0  },
                { 37, 28, 65.0, 130.0 },
                { 44, 28, 72.0, 144.0 },
                { 12, 29, 41.0, 82.0  },
                { 37, 29, 66.0, 132.0 },
                { 44, 29, 73.0, 146.0 },
                { 12, 30, 42.0, 84.0  },
                { 37, 30, 67.0, 134.0 },
                { 44, 30, 74.0, 148.0 }
        };
}
```

In questo caso le prime due colonne sono i due parametri combinati fra loro e le seconde due colonne i valori attesi per ogni coppia di parametri. Il modello CitLab da cui è stato ottenuto questo Data Provider contiene le seguenti

istruzioni:

```
Parameters:
        Numbers primo { 12 37 44 };
        Range secondo [ 25 .. 30 ];
end
ExpectedResults:
        sum checks ( (primo+textLines) );
        double_sum checks ( ((primo+textLines)*2) ) ;
end
```

`Primo` e `secondo` sono i due parametri definiti nel modello. Il primo assume i valori 12, 37 e 44 e il secondo un range di valori interi inclusi nell'intervallo 25 a 30 compresi.

Per ogni loro combinazione si valutano i valori attesi `sum` e `double_sum`.

Il valore atteso `sum` restituisce la somma di primo e secondo e `double_sum` la doppia somma di primo e secondo.

Terminato l'inserimento del codice relativo al metodo `rangeData()` si è passati alla creazione del metodo che ottiene in input il Data Provider. Questo metodo assume come nome quello scelto dal programmatore che si ottiene dal file `options.json`. È annotato con `@Test` per indicare a TestNG che esso è il metodo da testare ed ha come parametro il nome del Data Provider da utilizzare, corrispondente a quello appena descritto.

Il metodo ha in input i parametri dell'i-esimo caso di Test della Test Suite e dell'i-esima lista di valori attesi e l'insieme composto dall'unione di queste due i-esime liste di valori corrisponde al vettore restituito dal Data Provider all'i-esima chiamata.

È stato inoltre aggiunta una istruzione per stampare in console i parametri che si valutano ed i rispettivi valori attesi all'esecuzione dell'i-esimo Test della Test Suite.

Il codice prodotto nel file `.java` è simile al seguente, con il numero di parametri ed i loro nome differenti per ogni modello CitLab.

```
public void TestNumero1(Object primo, Object textLines,
    Object sum, String doubleSum) {
        System.out.println("primo: " + primo + ",
            textLines: " + textLines + " --> sum: " + sum
            + ", doubleSum: " + doubleSum);
        /* Add your asserts here */
}
```

In questo metodo di test lo sviluppatore inserirà le opportune istruzioni Assert per constatare la correttezza dell'implementazione dei propri metodi Le ultime istruzioni contenute nel metodo `generateOutput()` sono relative alla chiusura degli stream aperti nella fase iniziale relativi alla scrittura su file.

4.5 Piattaforma di Collaborazione

Il progetto di CitLab, incluso lo sviluppo di questo plugin, è completamente mantenuto in un repository remoto su SourceForge [13]. SourceForge è una piattaforma che consente lo sviluppo collaborativo di progetti fornendo la gestione del repository centralizzato. Offre un frontend per la gestione del progetto via web e molti IDE, tra cui Eclipse, consentono una semplice configurazione di repository remote SVN, come SourceForge, per eseguire tutte le operazioni di visualizzazione, aggiornamento, modifica e confronto delle versioni sul repository stesso.

5 Conclusioni

Il progetto è stato portato a termine nella sua totalità. Tutti gli obiettivi inizialmente prefissati sono stati raggiunti.
Come sviluppatore di TNG Exporter e autore di questa tesi esprimo contentezza e soddisfazione nell'aver contribuito allo sviluppo di CitLab.

Riferimenti bibliografici

[1] Software Testing Services: Market Research Report http://www.technavio.com/report/global-it-professional-services-global-software-testing-market-2017-2021.

[2] Combinatorial testing can detect hard-to-find software faults more efficiently than manual test case selection methods: http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-kacker-lei-hunter09.pdf

[3] Combinatorial Testing: Tools available (http://www.pairwise.org/tools.asp)

[4] Failure modes in medical device software: an analysis of 15 years of recall data (http://csrc.nist.gov/staff/Kuhn/final-rqse.pdf)

[5] IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays: (http://cs.unibg.it/gargantini/research/papers/amost09.pdf)

[6] Simulated Annealing (SA) (http://www1.unipa.it/valerio.lacagnina/pub/ricercaOperativa/SimulatedAnnealing.pdf)

[7] Junit Class Assert (http://junit.org/junit4/javadoc/latest/org/junit/Assert.html)

[8] Extensions and Extensions Points: https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points

[9] Junit 4 Vs TestNG: Comparison (https://www.mkyong.com/unittest/junit-4-vs-testng-comparison)

[10] JUnit vs TestNG: Which Testing Framework Should You Choose? (http://blog.takipi.com/junit-vs-testng-which-testing-framework-should-you-choose/)

[11] JSON standard (http://www.json.org/)

[12] Gson: the library (https://github.com/google/gson)

[13] SourceForce (https://sourceforge.net/)

[14] IPO-s: Incremental Generation of Combinatorial Interaction Test Data Based on Symmetries of Covering Arrays: https://www.computer.org/csdl/proceedings/icstw/2009/3671/00/3671a010-abs.html