

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Fondamenti di Informatica T-2

TEST DI ACCETTAZIONE PER

SCENARI MULTI-PARADIGMA:

STRUMENTI A CONFRONTO

CANDIDATO:

Marcello Colameo

RELATORE:

Chiar.mo Prof. Enrico Denti

CORRELATRICE:

Ing. Roberta Calegari

Anno Accademico 2013/2014

Sessione III

Indice

1. INTRODUZIONE.....	4
2. BACKGROUND.....	6
2.1. Testing Applicazioni	6
2.1.1. Tipologie di Collaudo	6
2.2. tuProlog.....	13
2.2.1. Test in tuProlog	14
3. ANALISI DEL PROBLEMA.....	16
3.1. Test di accettazione: requisiti	16
3.2. Test di accettazione in tuProlog: limiti	18
4. CONFRONTO DEGLI STRUMENTI.....	19
4.1. FIT	20
4.1.1. Caratteristiche Generali	20
4.1.2. Sito Web, Documentazione, Materiale reperibile su Internet	22
4.1.3. Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità	22
4.1.4. Facilità installazione & Integrazione Eclipse.....	23
4.1.5. Compatibilità con le Versioni Java.....	23
4.1.6. Facilità scrittura & formato Test	23
4.1.7. Estensibilità	24
4.2. FITNESS.....	24
4.2.1. Caratteristiche Generali	24
4.2.2. Sito Web, Documentazione, Materiale reperibile su Internet:	28
4.2.3. Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità:	28
4.2.4. Facilità installazione & Integrazione Eclipse:.....	28
4.2.5. Compatibilità con le Versioni Java:.....	29
4.2.6. Facilità scrittura & formato Test:	29
4.2.7. Estensibilità:	29
4.3. CONCORDION	29
4.3.1. Caratteristiche Generali	29
4.3.2. Sito Web, Documentazione, Materiale reperibile su Internet:	32
4.3.3. Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità:	32
4.3.4. Facilità installazione & Integrazione Eclipse:.....	32
4.3.5. Compatibilità con le Versioni Java:.....	33
4.3.6. Facilità scrittura & formato Test:	33
4.3.7. Estensibilità:	33
4.4. Confronto e scelta dello Strumento	33

5. CASO DI STUDIO: TUPROLOG	36
5.1. Progettazione Test di accettazione.....	36
5.1.1. Singleton.....	37
5.2. Implementazione Test di accettazione	39
5.2.1. Scrittura classi Fixtures.....	39
5.2.2. Scrittura Test XHTML.....	48
5.3. Collaudo	53
6. CONCLUSIONI	56

1. Introduzione

Ad oggi è sempre più in aumento la complessità del software, e non sempre le problematiche possono essere risolte nel modo più semplice ed efficiente, utilizzando un solo paradigma di programmazione. La soluzione è quella di utilizzare un linguaggio di programmazione multi paradigma. Un linguaggio di programmazione multi paradigma è un linguaggio di programmazione che supporta più di un paradigma di programmazione. L'idea centrale è quella di fornire un quadro in cui i programmatori possano lavorare in una varietà di stili, utilizzando i costrutti da diverse tipologie di paradigmi. L'obiettivo è quello quindi di permettere ai programmatori di utilizzare lo strumento migliore per lo sviluppo di software. In informatica, il collaudo è un procedimento che fa parte del ciclo di vita del software, utilizzato per individuare le carenze di correttezza, completezza e affidabilità delle componenti dell'applicazione in corso di sviluppo. Consiste nell'eseguire il software da collaudare, da solo o in combinazione ad altri programmi di servizio, e nel valutare se il comportamento rispetta i requisiti o meno. Tra le tipologie di Testing più utilizzate vi sono i Test di unità, come quelli effettuati attraverso l'utilizzo del Framework JUnit. Un'altra tipologia di collaudo è quella dei Test di accettazione. Uno dei Framework più utilizzati per questa tipologia di Test è FIT, uno strumento classico per l'ambiente Java, che però risale a qualche anno fa e soffre di difetti legati all'età.

Il primo obiettivo di questa tesi è quello di svolgere un confronto comparato degli strumenti ad oggi esistenti sul mercato per la scrittura di Test di accettazione, per arrivare alla selezione dello strumento ritenuto più idoneo con particolare riferimento alle applicazioni multi paradigma.

La selezione dello strumento avverrà attraverso tre fasi principali:

- ricerca di tutti i possibili strumenti per la scrittura dei Test di accettazione
- riduzione della lista degli strumenti in base alla valutazione di determinati parametri

- confronto approfondito degli strumenti rimasti per individuare lo strumento ritenuto più idoneo

Secondo obiettivo della tesi è quello di validare la scelta effettuata in uno scenario applicativo reale, ovvero in una complessa applicazione multi paradigma. Il progetto selezionato è tuProlog(un interprete Prolog scritto in Java), sviluppato presso il Dipartimento di Elettronica Informatica e Sistemistica della Facoltà di Ingegneria dell'Università degli Studi di Bologna. La riscrittura dei Test dovrà essere fatta attraverso una unione tra gli attuali Test FIT presenti, e quelli standard.

La tesi si svilupperà quindi come segue:

- nel Capitolo 2 verrà introdotto il problema del Testing delle applicazioni e descritto il software multi paradigma di cui terremo conto, ovvero tuProlog
- nel Capitolo 3 verrà effettuata un'analisi del problema, specificando i requisiti che vogliamo che siano presenti nello strumento scelto
- nel Capitolo 4 verrà effettuato un confronto comparato tra alcuni strumenti individuati su Internet. Al termine attraverso un confronto approfondito sceglieremo lo strumento
- nel Capitolo 5 verrà affrontato il caso di studio tuProlog, attraverso una fase di progettazione dei Test e successivamente una fase di implementazione
- nel Capitolo 6 verrà descritta la parte relativa alla fase di collaudo

2. Background

Poiché lo scopo di questa tesi consiste nell'analizzare gli strumenti e gli approcci di Testing in scenari multi-paradigma, si forniscono alcune nozioni di background per definire il contesto di lavoro. In particolare nel paragrafo 2.1 si forniscono le nozioni basi del problema di Testing applicativo con classificazione dei vari Test esistenti. Nel paragrafo 2.2 si introduce invece il sistema tuProlog, multi-paradigma e multi-piattaforma dalla nascita, che per le sue caratteristiche rappresenta un ottimo sistema su cui applicare lo studio eseguito.

2.1. Testing Applicazioni

Una delle fasi di progettazione di un Software consiste nel Testing dell'applicazione. L'attività di collaudo non è una attività accessoria, ma va accuratamente progettata ed è parte integrante del progetto. Progettando il collaudo a priori si possono ottenere diversi benefici verificando:

- la correttezza del Software: data una definizione dei requisiti che il Software deve soddisfare, il Software si dice corretto se rispetta tali requisiti
- la robustezza del Software: il Software si dice robusto se si comporta in maniera accettabile anche in corrispondenza di situazioni anomale.
- l'efficienza del Software: buon utilizzo delle risorse hardware
- l'integrità del Software: abilità del sistema Software di proteggere i vari componenti da accessi e/o modifiche non autorizzati di natura sia volontaria che involontaria

Il Testing è un processo di esecuzione del Software allo scopo di scoprirne i malfunzionamenti. Osservando i malfunzionamenti possiamo dedurre la presenza di difetti. Il Testing non può dimostrare l'assenza di difetti (e dunque la correttezza del programma), ma può solo dimostrare la presenza di difetti. Il processo di Testing deve mostrare che esso è pronto per l'uso operativo.

L'obiettivo del Testing è quindi quello di aiutare a localizzare i difetti e facilitare il debugging. La bontà dei Test si può misurare in termini di efficacia (numero di malfunzionamenti trovati / numero di malfunzionamenti da trovare) e efficienza (numero di Test in grado di scoprire malfunzionamenti / numero di Test totali).

2.1.1. Tipologie di Collaudo

Esistono varie tipologie di Testing: whitebox e blackbox. Nel caso del whitebox il collaudo è progettato e svolto tenendo esplicitamente in conto di come è fatto dentro un componente. Lo scopo è quello di verificare la corretta implementazione delle varie funzioni, stressandole ad una ad una anche nei casi critici. Poiché richiede la conoscenza e la comprensione della struttura interna del Software sotto Test, questa forma di Testing è in genere a carico di un programmatore spesso lo stesso che ha scritto il codice. E' una tecnica logic-driven e richiede la conoscenza della logica del programma e perciò utilizzata dai programmatori che hanno sviluppato il codice, si tratta del Test unitario (Unit Test), si Testano i percorsi logici del programma, i flussi primari e quelli secondari, le condizioni di selezione, le combinazioni particolari. Si utilizzano strumenti appositi di sviluppo (debugger) che permettono di agire direttamente sul codice sorgente: eseguire il codice partendo da istruzioni selezionate, fermare l'esecuzione in punti precisi e farla ripartire, visualizzare il contenuto dei campi e modificarlo a piacimento. Il debugging rappresenta la tecnica per eccellenza utilizzata dagli sviluppatori per correggere gli errori(bug). L'attività di debugging può risultare anche molto difficile a seconda della complessità del Software, della facilità di comprensione del codice, della possibilità di riprodurre le condizioni di errore, etc. Il processo di debugging prevede: analisi del problema e identificazione della causa del malfunzionamento, individuazione del componente in errore che ha generato il malfunzionamento, progettazione e implementazione della correzione dell'errore, Testing della correzione apportata e verifica dell'eliminazione del malfunzionamento. Per Unit Testing si intende l'attività di Testing di singole unità Software. Per unità si intende normalmente il minimo componente di un programma dotato di funzionamento autonomo; a

seconda del paradigma di programmazione o linguaggio di programmazione, questo può corrispondere per esempio a una singola funzione nella programmazione procedurale, o una singola classe o un singolo metodo nella programmazione a oggetti. Un esempio pratico di Unit Testing è JUnit, uno degli strumenti più utilizzati in ambiente Java. Il Test di unità può essere sviluppato in maniera indipendente dallo sviluppatore, ma è buona norma avere un prodotto standard e aderire ad una pratica comune. L'idea dello Unit Test in Java è quella di valutare ogni singolo metodo in funzione dei valori attesi. Ipotizzando la situazione in cui vi sia una funzione matematica che converta un numero positivo in un numero negativo, un caso di Test (Test case) verificherà che, quando passiamo il valore 5, il risultato sia -5. Se il Test ha esito negativo evidentemente c'è un errore da correggere. A partire dalla versione 4, JUnit, si avvale delle novità apportate da Java 5 (annotations), per rendere lo sviluppo di Test suite semplice ed immediato. Il processo di esecuzione è guidato dalle annotazioni:

Annotazione	Utilizzo
@Test	Per annotare i metodi di Test
@Before	Per annotare un metodo da eseguire prima dell'esecuzione di un Test case (ad esempio l'apertura di una connessione ad un database, o uno stream)
@After	Per annotare un metodo da eseguire dopo l'esecuzione di un Test case (chiusura di risorse aperte in precedenza)

@BeforeClass	Come @Before ma solo all'inizio dell'esecuzione del Test
@AfterClass	Come @After ma solo alla fine dell'esecuzione del Test
@Ignore	Utilizzata per evitare l'esecuzione di un Test (evitando di commentare)

Un esempio di Test

@Test

```
public void Test1() {

    System.out.println("Test 1");
    assertTrue(true);
    assertFalse(false);

}
```

I metodi Test (annotati opportunamente dall'annotazione @Test) verranno eseguiti senza un ordine preciso, quindi non necessariamente vengono eseguiti nell'ordine definito. Un Test fallisce se fallisce almeno uno dei metodi assert contenuti in esso. I metodi assert sono metodi statici contenuti nella classe `org.junit.Assert` (abbiamo effettuato un import statico in Testa alla classe) che effettuano un semplice confronto tra il risultato atteso ed il risultato dell'esecuzione. In questo caso non stiamo Testando nulla di concreto, semplicemente utilizziamo gli `assertTrue` e `assertFalse` su dei valori statici. Un Test potrebbe sollevare una eccezione.

Quindi Testing della robustezza di un'unità significa forzatamente sollevare un'eccezione. In questi casi è interessante l'uso del parametro "expected" nell'annotazione, dove indichiamo che è attesa l'eccezione `java.lang.Exception`. Se il metodo non solleva quel tipo di eccezione il Test sarà fallito.

Nel caso blackbox Testing il collaudo è progettato e svolto "ai morsetti" di un componente Software, senza sapere come è fatto dentro. Lo scopo è quello di verificare la correttezza del comportamento esterno, indipendentemente dalla realizzazione. Il collaudo effettuato accedendo al Software solamente tramite l'interfaccia utente, oppure tramite interfacce di comunicazione tra processi, viene detto "collaudo a scatola nera". Può essere manuale oppure automatizzato, e tipicamente è utilizzato per il collaudo di sistema, in quanto per collaudare l'intero sistema normalmente non è necessario richiamare singole routine. Se si tratta di un collaudo automatizzato, le routine di collaudo sono prevalentemente scritte in un linguaggio di livello più alto del linguaggio con cui è stata scritta l'applicazione, a volte in un linguaggio progettato appositamente per il collaudo. Il Software da collaudare e il Software di collaudo costituiscono processi distinti comunicanti. Per il collaudo a scatola nera manuale non è richiesta l'opera di un programmatore, e per quello automatizzato è sufficiente un programmatore con modeste competenze. Spesso il collaudo a scatola nera è effettuato da persone che non fanno parte dell'organizzazione che sviluppa il Software; si tratta degli utilizzatori stessi che effettuano il collaudo Beta. Un vantaggio del collaudo a scatola nera sta, nei casi in cui il codice sorgente e la documentazione di progetto sono segreti industriali, nel fatto che tale collaudo può essere effettuato anche da persone esterne all'azienda. Un altro vantaggio sta nel fatto che per tale tipo di collaudo non sono necessari programmatori esperti e che conoscano gli aspetti interni del Software da collaudare. Pertanto, si possono trovare molti più collaudatori, senza dover investire nell'addestramento. La tecnica "Black-box" permette di verificare il comportamento esterno del Software. E' governata dall'immissione dei dati in ingresso (Data-driven), il Software è esercitato

tramite i casi di prova progettati in base alle funzionalità richieste (requisiti funzionali) e in base alle caratteristiche strutturali richieste (requisiti non funzionali) come l'usabilità, le performance, il carico di lavoro capace di supportare, etc. La tipologia di Test Blackbox più utilizzata sono quelli dei Test di accettazione. I Test di accettazione servono a dimostrare la correttezza dei requisiti funzionali di un Software o di un intero sistema. Il cliente può accettare o meno il prodotto Software richiesto in base a dei propri e ben precisi criteri di accettazione. I Test di accettazione sono delle specifiche per descrivere il comportamento e le funzionalità attese di un sistema. I Test di accettazione ci dicono, data una storia utente, come il sistema elabora gli input in alcune condizioni e quali sono gli output di questa elaborazione. Le storie utente rappresentano un modo estremamente semplice per esprimere i requisiti. Sulla base delle storie utente si progettano i Test di accettazione. (Immagine 2.1.1)

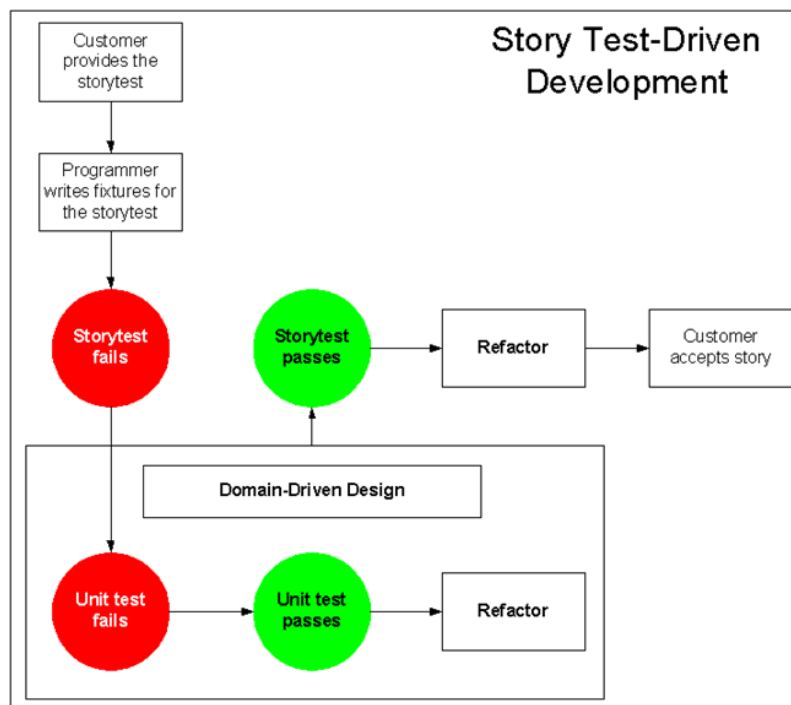


Figura 2.2.1: Fase sviluppo storie utente

Uno dei Framework più utilizzati per la scrittura dei Test di accettazione è FIT. Gli story Test sono tradotti sottoforma di tabelle (HTML, XML, Excel, ecc). Supporta la collaborazione tra

- Clienti e sviluppatori durante la fase di analisi
- Clienti e Tester durante la creazione degli story Test
- Sviluppatori e Tester durante l'implementazione delle fixtures
- Tester e clienti durante l'esecuzione degli story Test

Le FIT Tables migliorano l'espressione di quale sia il comportamento del sistema. FIT utilizza le tabelle per scrivere i Test e riportarne i risultati. In figura 2.2.2 è presente il flusso di sviluppo di FIT.

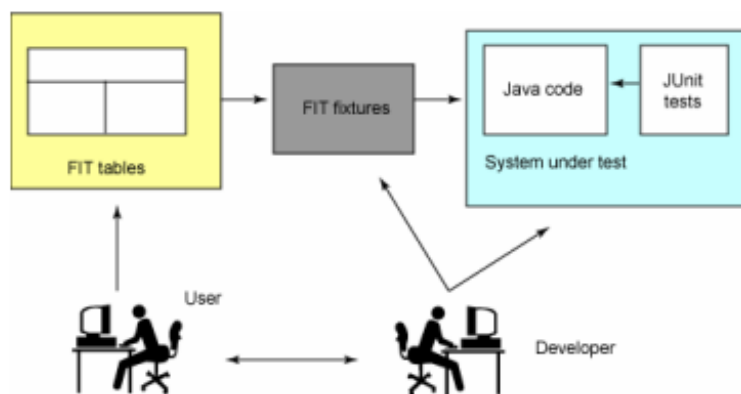


Figura 2.2.2: Flusso di sviluppo di FIT

2.2. tuProlog

tuProlog è un interprete Prolog, sviluppato in Java, .NET e iOS, presso il DISI della Facoltà di Ingegneria dell'Università degli Studi di Bologna. Le caratteristiche principali dell'interprete sono la leggerezza e la configurabilità. La configurabilità è ottenuta grazie ad un meccanismo che consente di caricare e scaricare all'interno di un motore tuProlog predicati, funtori e operatori, grazie al concetto di libreria. Alcune librerie sono incluse nella distribuzione standard di tuProlog, altre possono essere scritte dall'utente. tuProlog è fortemente integrato con Java, tanto da permettere l'utilizzo di oggetti Java all'interno di un'applicazione Prolog oppure l'utilizzo di un motore Prolog all'interno di applicazioni Java. Le librerie sono il mezzo attraverso il quale tuProlog realizza la sua caratteristica di configurabilità dinamica. Il motore è un nucleo minimale e leggero per scelta progettuale: include solo pochi predicati "built-in", definiti staticamente al suo interno. Ogni altra forma di funzionalità, come predicati, funtori, flag ed operatori, viene fornita dalle librerie, le quali possono essere aggiunte o rimosse dal motore dinamicamente. In particolare, tuProlog include all'interno della sua distribuzione delle librerie standard, che sono caricate automaticamente nel motore. L'integrazione con Java è completa e trasparente: utilizzando il linguaggio Prolog è possibile creare, ed utilizzare ogni entità Java, grazie alla libreria JavaLibrary. Utilizzando il linguaggio Java, è dualmente possibile invocare ed usare un motore tuProlog come un qualsiasi oggetto. Quindi, per esempio, è possibile da un lato utilizzare direttamente da Prolog qualunque package Java come Swing, arricchendo, in questa maniera, l'interprete tuProlog della capacità di interagire con componenti grafici; dall'altro lato è possibile sfruttare dall'ambiente Java più motori tuProlog, eventualmente configurati in maniera diversa, ognuno con le proprie librerie.

2.2.1.Test in tuProlog

L'interprete tuProlog è stato sottoposto a tre tipologie di Test:

- Test di unità
- Test di accettazione
- Test delle prestazioni

Di seguito verranno forniti i dettagli di ognuna delle categorie.

2.2.1.1. Test unità

L'obiettivo primario dei Test di unità consiste nell'isolare la parte più piccola di Software Testabile nell'applicazione dal resto del codice per stabilire se funziona esattamente come previsto. Ogni singola unità viene sottoposta a Test prima di essere successivamente integrata in moduli più complessi. I Test di unità per tuProlog prevedono il controllo del corretto funzionamento delle strutture dati di base dell'interprete, ossia le classi che derivano da Term, come Struct, Var e quelle numeriche (e.g. Int e Double), dei principali elementi del motore Prolog e di conseguenza del Parser, del gestore delle teorie, delle classi StructEnumerator e TermEnumerator e della principali librerie. Per effettuare i Test di unità sulla versione Java è stato utilizzato il Framework JUnit 4.8.2 che agevola la stesura e l'esecuzione dei Test.

2.2.1.2. Test accettazione

I Test di accettazione fanno parte di quella categoria di Testing detta "Black-Box Testing" in cui il programma è considerato come una scatola nera ed è Testato nella sua interezza, osservando la relazione "input-output". Questi Test non si basano quindi sulla struttura interna del sistema da

collaudare ma piuttosto sulle specifiche di come deve funzionare. Per quanto riguarda tuProlog, i Test di accettazione Testano tutti i predicati e funtori built-in e quelli messi a disposizione dalle librerie standard. Il Framework adottato per svolgere questo tipo di Test è FIT (Framework for Integrated Test). L'utente fornisce degli esempi di corretto funzionamento del sistema formattati in tabelle e salvati in un documento HTML. Tali esempi vengono connessi dal programmatore al Software da collaudare tramite le fixture. Durante l'esecuzione dei Test viene controllato il documento, e creata una copia in cui: i Test che hanno avuto successo sono colorati di verde, quelli che hanno lanciato eccezione di giallo e quelli che hanno fallito in rosso. Per automatizzare l'esecuzione dei Test FIT è stato utilizzato lo strumento GUIRunner. Tale strumento permette di eseguire in sequenza diversi Test in automatico e ne mostra i risultati in un'unica tabella seguendo lo stesso schema di colori utilizzato nelle tabelle FIT (verde, giallo, rosso).

2.2.1.3. Test prestazioni

I Test delle prestazioni sono stati realizzati per verificare la velocità di elaborazione dell'interprete. L'obiettivo è quindi solo quello di confrontare le prestazioni, la correttezza del comportamento viene data per scontata. Lo schema di esecuzione è il seguente: viene letta da file una particolare teoria, viene stabilita l'obiettivo da risolvere ed infine viene calcolato il tempo medio di esecuzione come differenza tra l'istante in cui inizia l'elaborazione e quello in cui finisce, diviso il numero di ripetizioni dell'operazione. Le ripetizioni per ogni Test sono state 700. Ogni Test è costituito da una teoria (file *.pro*) e da una coppia di classi (Java, C#) che si occupano di lanciare e misurare i tempi di risposta del Test sulle due diverse piattaforme.

3. Analisi del Problema

L'obiettivo principale di questo lavoro è fornire uno screening ed una analisi critica degli strumenti per la scrittura dei Test di accettazione oggi presenti sul mercato, individuandone punti di forza e debolezze, al fine di selezionare lo strumento migliore. A questo fine si definiscono i nuovi requisiti, paragrafo 3.1, che dovrà soddisfare lo strumento, introducendo una griglia di valutazione che ne consenta una valutazione numerica. Una volta selezionato il miglior strumento, se ne vuole verificare l'effettiva efficacia in scenari applicativi complessi, con particolare enfasi su applicazioni multi paradigma. Per questo, selezionato come caso di studio tuProlog, se ne vuole rinnovare la Suite di Test di accettazione. Nel paragrafo 3.2 si dettaglierà l'analisi degli attuali Test di accettazione presenti in tuProlog enfatizzandone i limiti e le criticità, anche rispetto ai requisiti definiti. Nel capitolo 4 verranno selezionati i Software più rilevanti e verranno valutati in base ai criteri definiti al fine di avere una valutazione complessiva di ognuno.

3.1. Test di accettazione: requisiti

L'obiettivo è quindi quello di effettuare un confronto comparato di una serie di strumenti per la scrittura dei Test di accettazione, in particolare riferiti ad una applicazione multi paradigma.

I requisiti di cui bisognerà tener conto nella valutazione degli strumenti sono i seguenti:

- Ricerca di un strumento Open Source: questo è un requisito molto importante soprattutto se si vuole valutare di modificare lo strumento e/o di estenderlo
- Integrazione JUnit: in modo da semplificare e velocizzare la fase di scrittura dei Test
- Compatibilità con le ultime versioni di Java: in particolare la 8, dato che le applicazioni multi paradigma si basano molto su quest'ultima versione

- Progetto supportato: per non rischiare di avere in futuro uno strumento obsoleto
- Portabilità e linguaggi supportati: dato che in futuro potrebbe essere necessario la scrittura dei Test in qualche altro linguaggio
- Semplicità e chiarezza nella scrittura dei Test: se i Test si importano/scrivono in modo accettabile o vanno invece riscritti ad hoc in qualche formato proprietario
- Formato dei dati in output: se i risultati dell'esecuzione sono presentati in modo chiaro, in un formato open, magari facilmente modificabili rispetto alle impostazioni a default o importabili attraverso l'utilizzo di altri strumenti
- Facilità di installazione
- Integrazione con gli editor di Programmazione (Eclipse, NetBeans, ecc)
- Tempo di esecuzione dei Test
- Documentazione
- Semplicità di apprendimento

3.2. Test di accettazione in tuProlog: limiti

In tuProlog per la scrittura dei Test di accettazione è stato utilizzato FIT: FIT è uno strumento classico per l'ambiente Java, che però risale a qualche anno fa e soffre di difetti legati all'età, tra cui lo scarso supporto ai tipi generici.

Tra le principali problematiche nell'utilizzo di FIT, abbiamo una gestione abbastanza complessa dei Test e il fatto che molte caratteristiche presenti negli strumenti attuali non sono disponibili. Per l'esecuzione dei Test di accettazione sono necessari degli strumenti esterni e spesso questi strumenti non si integrano perfettamente con i vari editor utilizzati durante la programmazione.

In FIT il codice Java è abbastanza verboso, e il codice HTML risulta poco chiaro durante la scrittura e vi è troppa dipendenza tra i due linguaggi. Un altro problema è legato alla gestione delle eccezioni, infatti FIT non le gestisce perfettamente, e spesso capita che vengano notificate erroneamente.

Il Framework non ha perfetta gestione della tipizzazione dei dati, infatti in un semplice confronto (Equals), viene effettuato un'uguaglianza tra due stringhe, senza tener conto del tipo di dato. Inoltre FIT non presenta compatibilità con Java 8 e vi sono problemi anche con Java 7 nella gestione dei tipi generici, infatti in alcuni casi per effettuare dei confronti sarebbe molto più comodo utilizzare delle Liste e/o Mappe e con un semplice metodo verificare se quel determinato parametro è presente nella lista. Inoltre è difficilmente estensibile, ed il set di istruzioni è molto ridotto. Data l'età di FIT, esso non è più supportato ed è anche poco adatto ad eventuali estensioni.

4. Confronto degli strumenti

Il primo obiettivo di questa tesi è stato quello di trovare un Framework per la scrittura di Test di accettazione, cercando di considerare quello che potrebbe essere individuato come il miglior strumento attualmente esistente. Il secondo passo sarà poi quello di riscrivere i Test attraverso l'utilizzo del Framework scelto. Inizialmente è stata effettuata una ricerca su Internet per individuare una lista contenente tutti gli strumenti utilizzati per la scrittura dei Test di accettazione. Successivamente tenendo conto di determinati parametri si è deciso di scremare il numero di strumenti, fino ad arrivare a due possibili successori di FIT: Concordion e FITNESSE. I parametri di cui si è tenuto conto per la "scrematura" sono stati quelli presentati nel paragrafo 3.1.

Si è arrivati alla scelta dei due strumenti precedentemente indicati, dato che gli altri non soddisfacevano tutti i parametri o parte di essi.

Si è deciso quindi di effettuare un confronto molto approfondito per vedere quale si adattasse meglio alle nostre esigenze. I parametri che sono stati utilizzati per effettuare il confronto sono stati i seguenti:

- Caratteristiche Generali
- Sito Web, Documentazione, Materiale reperibile su Internet
- Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità
- Facilità installazione & Integrazione Eclipse
- Compatibilità con le Versioni Java
- Facilità scrittura & formato Test
- Estensibilità

Nei paragrafi successivi verranno prese in esame le caratteristiche sopra citate per ognuno degli strumenti selezionati, al fine di approfondirne gli aspetti. Per completezza, e al fine di avere il background di confronto nel paragrafo 4.1 si riportano le stesse caratteristiche relative al Framework FIT.

4.1. FIT

4.1.1. Caratteristiche Generali

FIT(Framework for Integrated Test) è un Framework che automatizza i Test di accettazione. Gli History Test possono essere scritti in vari formati, ma principalmente in HTML.

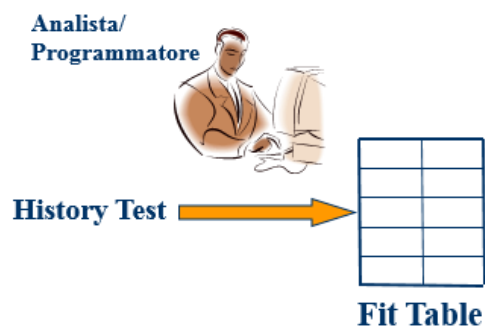


Figura 4.1.1: Struttura alto livello di FIT

A più alto livello abbiamo le Fit Tables (Figura 4.1.1). FIT utilizza le tabelle per scrivere i Test, per riportarne i risultati e per migliorare la comunicazione tra i vari analisti di quale debba essere il comportamento del sistema. Una tabella FIT è un modo di esprimere la logica di business utilizzando una semplice tabella HTML. Le tabelle possono essere create anche attraverso l'utilizzo di Editor esterni, anche senza scrivere una riga di codice, quindi per la loro scrittura non è necessario

un programmatore. Abbiamo poi le Fixtures, una per ogni tabella, responsabile di verificare che i Test nelle relative tabelle siano stati soddisfatti dal sistema sotto Test, ovvero coloro che si occupano dell'elaborazione vera e propria.

A più basso livello abbiamo il sistema vero e proprio da collaudare. (Figura 4.1.2)

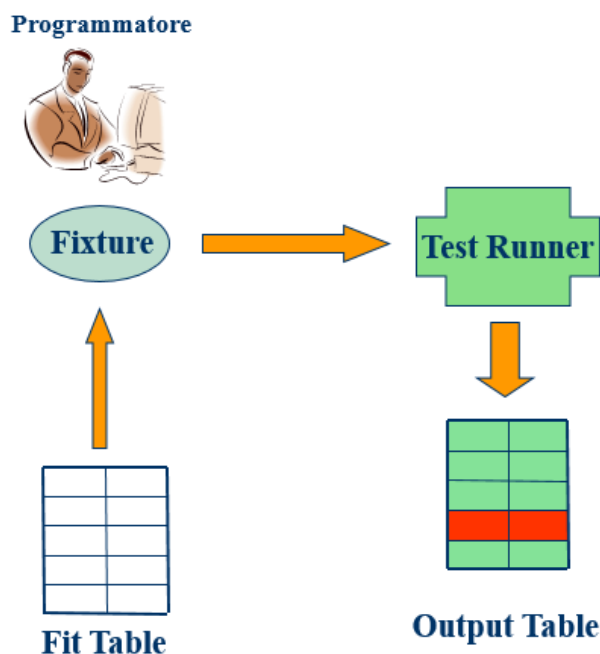


Figura 4.1.2: Struttura basso livello di FIT

Esistono varie tipologie di Fixture: Column Fixture, Action Fixture, Row Fixture. Le ColumnFixtures Tables sono utilizzate essenzialmente per collaudare dei calcoli ottenuti seguendo alcune regole di business. Le Column Fixtures usano la metafora del semaforo per indicare lo stato dei Test: Verde: Test superato, Rosso: Test non superato, Giallo: parte del Test non è completamente implementato, o qualcos'altro non è andato per il verso giusto, Grigio: esistono parti di tabelle che non possono essere state eseguite. Una ActionFixture Table Testa che una sequenza di

azioni eseguite su di un sistema abbiano gli effetti desiderati. Le ActionFixtures utilizzano la metafora del device per controllare i Test: Gli input possono essere inseriti in campi di Testo, i pulsanti possono essere cliccati, i campi di Output confrontati con valori attesi. Sono utilizzate per eseguire una serie di azioni e Test su un sistema che cambia di stato.

Nelle ActionFixtures Tables si possono eseguire quattro tipi di operazioni:

- **Start:** invocabile una sola volta, serve per settare i parametri iniziali
- **Enter:** per settare il valore di una variabile
- **Press:** per stampare il valore di una variabile
- **Check:** per effettuare un confronto sul valore di una variabile

Le RowFixtures Table sono utilizzate per collaudare che i risultati di una query o di una ricerca eseguita sul sistema sotto Test diano i risultati attesi. Usate anche per verificare che gli elementi attesi di listgroup, list, sequence, ecc siano presenti nel sistema sotto Test. A differenza delle ColumnFixture le righe sono tra loro dipendenti, quindi bisogna tener conto di una sorta di “parallelismo” durante l’esecuzione.

4.1.2. Sito Web, Documentazione, Materiale reperibile su Internet

Il Sito Web è molto semplice da utilizzare, ma non è più aggiornato ed è molto scarso riguardo la documentazione. Su Internet si trovano molti riferimenti a FIT, ma spesso più come menzioni storiche, infatti spesso è solo utilizzato per introdurre i suoi successori come FITNESSE e Conordion.

4.1.3. Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità

La licenza è di tipo GNU (General Public License). Il progetto non è più aggiornato. I Linguaggi supportati sono i seguenti: Java, .NET, Python, Perl, Smalltalk, C++, Ruby, Delphi. (Object C e Php non completamente supportati). Riguardo la portabilità, essendo scritto in Java, non ci sono problemi.

4.1.4.Facilità installazione & Integrazione Eclipse

Se si vuole utilizzare FIT senza l'ausilio di Eclipse, il suo utilizzo è abbastanza immediato. Una volta scaricato lo zip, avremo il file fit.jar, che dovrà essere invocato da riga di comando, passando come parametri il file .html contenente le tabelle di accettazione e il nome che dovrà avere il file in output una volta elaborato. Per velocizzare la loro scrittura ed esecuzione è possibile installare un Plugin(FitRunner), molto semplice da utilizzare, ma non compatibile con le ultime versioni di Eclipse. In definitiva quindi per una corretta ottimizzazione, bisognerà utilizzare un editor WEB, eventualmente anche grafico, il codice Java scritto con Eclipse e per ottenere i risultati utilizzare la riga di comando. Una delle poche note positive è che gli esempi allegati nel momento in cui si effettua il download del Framework, sono molto semplici e chiari e permettono un facile apprendimento sull'utilizzo dello strumento. Esistono degli strumenti che aiutano lo sviluppo dei FIT Test, che però si basano sempre sull'obsoleto FIT. In questo caso non sono stati testati approfonditamente, dato che il "core" è sempre quello base.

4.1.5.Compatibilità con le Versioni Java

L'ultima versione compatibile è la 1.3.

4.1.6.Facilità scrittura & formato Test

I Test devono essere scritti in HTML. Basandosi completamente sull'utilizzo di tabelle, è necessaria la conoscenza di 4 Tag. E' possibile però utilizzare degli editor grafici, che permettono la scrittura di pagine web senza scrivere codice HTML, questi strumenti sono molto comodi ma non sempre sono perfetti, ed è necessaria spesso quindi una successiva correzione mettendo mano al codice HTML. Nonostante FIT sia relativamente obsoleto, la scrittura dei Test è la più semplice possibile, superata solo dalla scrittura di Test attraverso l'utilizzo di tabelle Excel. In ogni tabella bisogna specificare nella prima riga la classe (compreso il package), della classe Fixture che dovrà elaborare quella tabella. Nella seconda riga vanno specificate le funzioni che verranno richiamate durante l'elaborazione, nelle righe successive i dati che verranno confrontati. I risultati, essendo in HTML,

sono facilmente importabili dai principali editor di Testo, anzi si potrebbe pensare di creare uno programma apposito per effettuare il parsing dei dati in output.

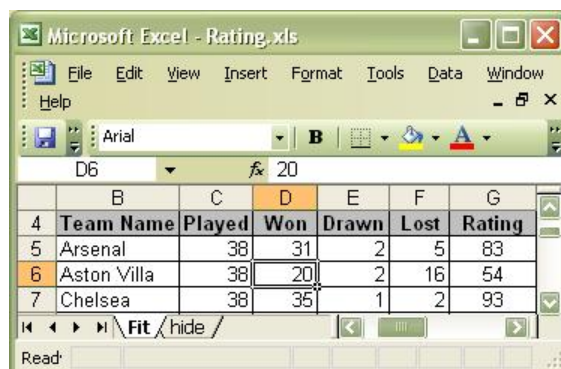
4.1.7. Estensibilità

Tutti i file Java si basano su una classe Fixture.java che deve essere implementata. Per effettuare delle estensioni, basta modificare la classe base. Essendo però molto obsoleto è poco conveniente pensare di estenderlo.

4.2. FITNESSE

4.2.1. Caratteristiche Generali

FITNESSE è un Framework che automatizza i Test di accettazione, nato come successo dell'obsoleto Framework FIT. Gli History Test possono essere scritti in vari formati, oltre HTML possono essere scritti in Excel (Figura 4.1.1), Word, ed altri formati proprietari che hanno messo a disposizione gli standard per poter implementare un apposito Plugin.



The screenshot shows a Microsoft Excel window titled "Rating.xls". The active cell is D6, containing the value 20. The table below is a football league table with columns: Team Name, Played, Won, Drawn, Lost, and Rating. The data rows are for Arsenal, Aston Villa, and Chelsea. The value 20 in cell D6 corresponds to the 'Won' column for Aston Villa.

	B	C	D	E	F	G
	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

Figura 4.1.1: Scrittura Test con Excel

La novità principale che è stata introdotta in FITNESSE è che vi è la possibilità di scrittura dei Test attraverso un'apposita WIKI, strumento molto potente, ma non di semplice apprendimento. Una Wiki è una collezione di documenti ipertestuali che permette a ciascuno dei suoi utilizzatori di aggiungere contenuti, come in un forum, ma anche di modificare i contenuti esistenti inseriti da altri utilizzatori.

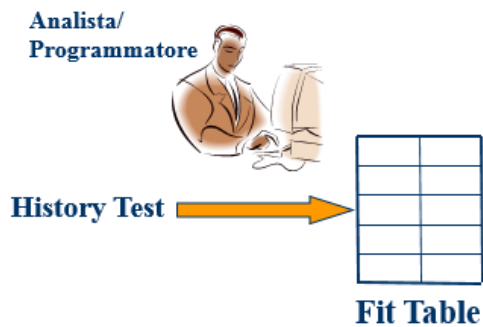


Figura 4.1.2: Struttura alto livello di FITNESSE

Come nel caso di FIT, a più alto livello abbiamo le Fit Tables (Figura 4.1.1). FITNESSE utilizza le tabelle per scrivere i Test, per riportarne i risultati e per migliorare la comunicazione tra i vari analisti di quale debba essere il comportamento del sistema. Le tabelle possono essere create anche attraverso l'utilizzo di Editor esterni, anche senza scrivere una riga di codice, quindi per la loro scrittura non è necessario un programmatore. Per la scrittura è possibile utilizzare anche la WIKI. Abbiamo poi le Fixtures, una per ogni tabella, responsabile di verificare che i Test nelle relative tabelle siano stati soddisfatti dal sistema sotto Test, ovvero coloro che si occupano dell'elaborazione vera e propria. La parte relativa alle Fixtures è rimasta invariata rispetto al FIT originale.

A più basso livello abbiamo il sistema vero e proprio da collaudare. (Figura 4.1.2)

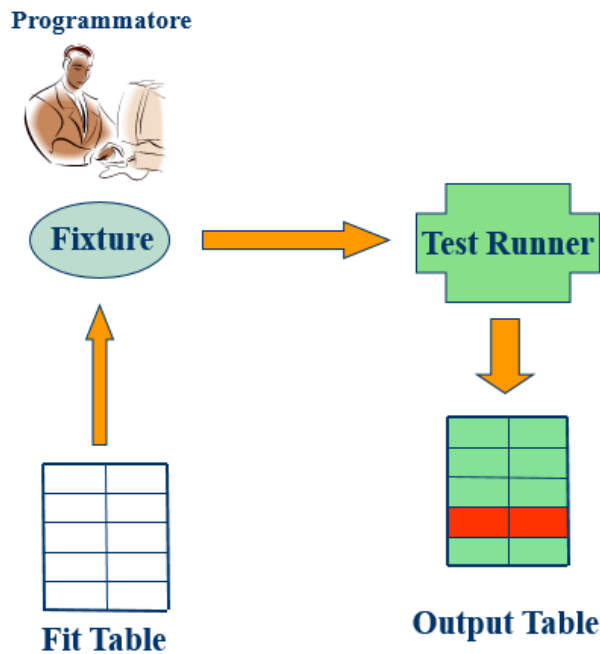


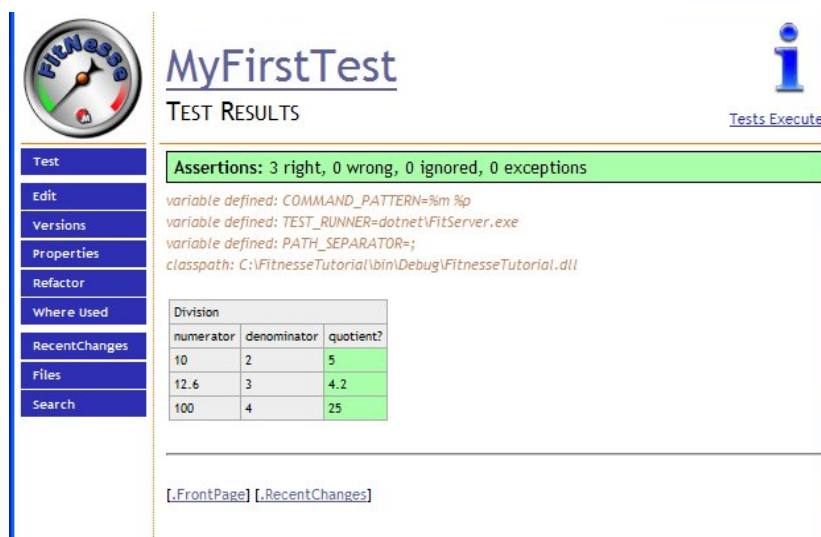
Figura 4.1.3: Struttura basso livello di FIT

Esistono varie tipologie di Fixture: Column Fixture, Action Fixture, Row Fixture. Le ColumnFixtures Tables sono utilizzate essenzialmente per collaudare dei calcoli ottenuti seguendo alcune regole di business. Le Column Fixtures usano la metafora del semaforo per indicare lo stato dei Test: Verde: Test superato, Rosso: Test non superato, Giallo: parte del Test non è completamente implementato, o qualcos'altro non è andato per il verso giusto, Grigio: esistono parti di tabelle che non possono essere state eseguite. Una ActionFixture Table Testa che una sequenza di azioni eseguite su di un sistema abbiano gli effetti desiderati. Le ActionFixtures utilizzano la metafora del device per controllare i Test: gli input possono essere inseriti in campi di Testo, i pulsanti possono essere cliccati, i campi di Output confrontati con valori attesi. Sono utilizzate per eseguire una serie di azioni e Test su un sistema che cambia di stato.

Nelle ActionFixtures Tables si possono eseguire quattro tipi di operazioni:

- **Start:** invocabile una sola volta, serve per settare i parametri iniziali
- **Enter:** per settare il valore di una variabile
- **Press:** per stampare il valore di una variabile
- **Check:** per effettuare un confronto sul valore di una variabile

Le RowFixtures Table sono utilizzate per collaudare che i risultati di una query o di una ricerca eseguita sul sistema sotto Test diano i risultati attesi. Usate anche per verificare che gli elementi attesi di listgroup, list, sequence, ecc siano presenti nel sistema sotto Test. A differenza delle ColumnFixture le righe sono tra loro dipendenti, quindi bisogna tener conto di una sorta di “parallelismo” durante l’esecuzione. Da come emerge dalle caratteristiche, FITNESSE non è altro che un FIT aggiornato con l’introduzione di una WIKI. Per utilizzare la WIKI bisogna installarla ed avviarla, definire il progetto dandogli il nome, scrivere le tabelle utilizzando la notazione specifica, scrivere il codice Fixture e collegarlo alla WIKI, eseguire i Test attraverso il tasto apposito presente sulla WIKI, e alla fine avremo in Output i risultati. (Figura 4.1.4)



The screenshot shows the 'MyFirstTest' application interface. On the left is a sidebar with a 'Fitness' logo and a menu with links: Test, Edit, Versions, Properties, Refactor, Where Used, RecentChanges, Files, and Search. The main content area is titled 'MyFirstTest TEST RESULTS' and features an information icon and a 'Tests Execute' link. A green banner displays the test status: 'Assertions: 3 right, 0 wrong, 0 ignored, 0 exceptions'. Below this, environment variables are listed: 'variable defined: COMMAND_PATTERN=%m %p', 'variable defined: TEST_RUNNER=dotnet\FitServer.exe', 'variable defined: PATH_SEPARATOR=;', and 'classpath: C:\FitnessTutorial\bin\Debug\FitnessTutorial.dll'. A table titled 'Division' shows the results of three test cases:

numerator	denominator	quotient?
10	2	5
12.6	3	4.2
100	4	25

At the bottom of the main content area, there are two links: '[.FrontPage]' and '[.RecentChanges]'.

Figura 4.1.4: Risultati Test FITNESSE

4.2.2.Sito Web, Documentazione, Materiale reperibile su Internet:

Il Sito Web è di difficile comprensione, e sono necessarie delle accurate ricerche per arrivare alla documentazione. Il sito è comunque molto aggiornato e presenta molta documentazione, anche troppa. Sicuramente uno dei suoi punti di forza è che è presente molto materiale su Internet, e c'è dietro una comunità di sviluppo molto attiva.

4.2.3.Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità:

La licenza è di tipo Apache License 2.0. Il progetto è aggiornato molto di frequente, anche se attualmente ha raggiunto una maturità tale per cui non sono necessari degli update frequenti. I Linguaggi supportati sono i seguenti: Java, .NET, Python, Perl, Smaltlak, C++, Ruby, Object C. Riguardo la portabilità, essendo scritto in Java, non ci sono problemi.

4.2.4.Facilità installazione & Integrazione Eclipse:

Se si vuole utilizzare FITNESSE senza l'ausilio di Eclipse, il suo utilizzo è abbastanza immediato. Una volta scaricato tutto il necessario dal sito web, bisognerà procedere all'installazione della WIKI. L'installazione della WIKI è un'operazione non immediata e spesso si possono verificare delle problematiche di compatibilità. Una volta installata, può essere avviata attraverso l'utilizzo del Browser. Se si utilizza Eclipse solo per la scrittura di codice Java bisognerà ricordarsi di effettuare un collegamento tra le tabelle WIKI e il codice Fixture, questo avviene specificando come primo parametro nelle tabelle, la classe che si dovrà occupare della sua elaborazione. Per l'esecuzione dei Test è necessario eseguire dalla WIKI il comando di Test ed otterremo, quindi, in Output i risultati. E' possibile installare un Plugin Eclipse, il quale ci permetterà di utilizzare la WIKI all'interno dell'editor e di poter verificare eventuali errori prima dell'esecuzione. Questo metodo è sicuramente il più consigliato, principalmente per questioni legate ai tempi e sia perché il collegamento tra Test e Fixtures avviene in modo automatico, cosa che spesso se fatta manualmente può trarre in inganno il programmatore.

4.2.5.Compatibilità con le Versioni Java:

L'ultima versione compatibile è la 7

4.2.6.Facilità scrittura & formato Test:

I Test possono essere scritti in HTML, ma di solito quando si utilizza FITNESSE si preferisce l'utilizzo della WIKI. E' possibile utilizzare degli editor esterni che rendono più agevole la scrittura delle tabelle. Questi strumenti sono molto comodi ma non sempre sono perfetti, ed è necessaria spesso quindi una successiva correzione mettendo mano al codice. La WIKI è sì uno strumento che può essere utilizzato da chi non sa programmare, ma spesso esso risulta molto rognoso per chi ha un minimo di conoscenza informatica. Infatti per il programmatore risulta molto più intuitivo la scrittura dei Test attraverso l'utilizzo del codice HTML. FITNESSE mette a disposizione un plugin sia per la scrittura dei Test attraverso l'utilizzo di tabelle Excel, ma anche la possibilità di esportarle nello stesso formato. Riguardo l'esportabilità dei dati è sicuramente uno dei migliori, dato che esistono moltissimi Plugin, che permettono di esportare i risultati in molti formati.

4.2.7.Estensibilità:

Tutti i file Java si basano su una classe Fixture.java che deve essere implementata. Per effettuare delle estensioni, basta modificare la classe base. FITNESSE mette a disposizione anche dei Plugin per rendere più agevole le modifiche al suo codice. Uno dei principali vantaggi di FITNESSE è che esistono un numero spropositato di Plugin, il che rende di rado necessario procedere ad eventuali estensioni.

4.3. CONCORDION

4.3.1.Caratteristiche Generali

Concordion è uno strumento Open Source per la scrittura di Test di accettazione, nato per risolvere i difetti dello strumento FITNESSE. I suoi principali vantaggi sono legati alla sua facilità di

apprendimento e di utilizzo. Il formato utilizzato per la scrittura dei Test di accettazione è l'XHTML. Per il suo utilizzo è necessario l'apprendimento di un piccolo set di comandi. Inoltre per l'esecuzione dei Test non è necessario l'installazione di un Software esterno, dato che Concordion si integra perfettamente con JUnit. Oltre la scrittura dei Test in XHTML, è necessario scrivere delle classi Java Fixture, ovvero quelle classi che si occuperanno dell'esecuzione dei Test e che restituiranno in output i risultati dei Test. Un vincolo di Concordion è che ad ogni file XHTML di Test, dovrà corrispondere un file .java con lo stesso nome. All'interno delle Fixture devono essere inseriti tutti i metodi utilizzati per l'esecuzione dei Test. In Figura 4.3.1 è possibile vedere la struttura completa di Concordion.

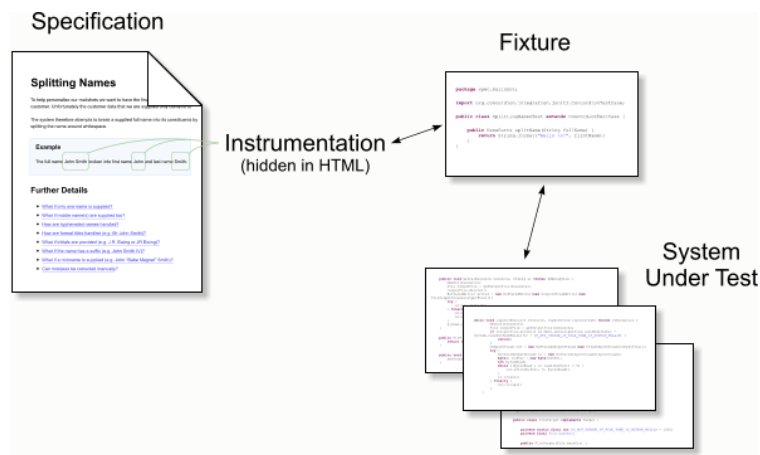


Figura 4.3.1: Struttura Concordion

Dal sito è possibile scaricare un progetto d'esempio. Nella figura 4.3.2 è possibile vedere i file presenti all'interno del kit scaricato.

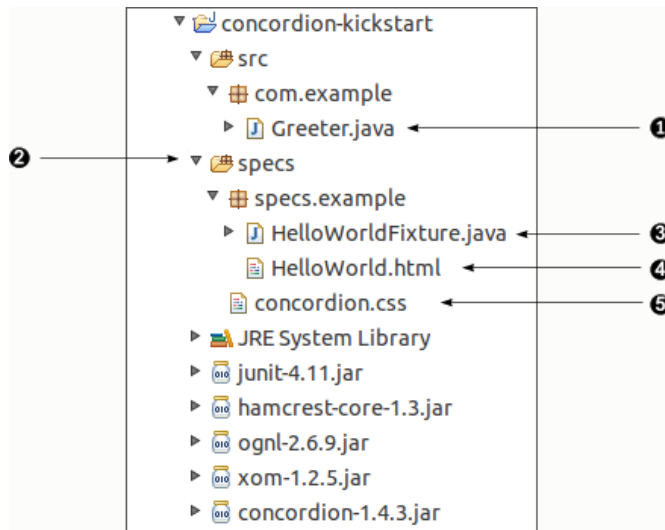


Figura 4.3.2: Struttura Concordion in Eclipse

- Greeter.java è la classe che vogliamo collaudare
- Specs è la cartella che contiene tutti i file relativi a Concordion
- HelloWorldFixture.java è la classe Fixture che contiene i metodi utilizzati nei Test
- HelloWorld.html è il file contenente i Test di accettazione
- Concordion.css è il foglio di stile dell'output prodotto da Concordion

Inoltre devono essere importati i .jar presenti in figura.

Nel file XHTML bisogna specificare che utilizzeremo il Namespace Concordion

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
```

Il comando più utilizzato è l'assertEquals , come nell'esempio sottostante

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
  <body>
    <p concordion:assertEquals="getGreeting()">Hello World!</p>
  </body>
</html>
```

Nel file Fixture bisogna poi specificare il metodo `getGreeting()`;

Principali comandi di Concordion:

- `assertEquals`: effettua un confronto del valore con il risultato restituito da un metodo
- `assertTrue`: verifica che il risultato del metodo sia `True`
- `assertFalse`: verifica che il risultato del metodo sia `False`
- `execute`: permette l'esecuzione di un metodo presente all'interno delle Fixture
- `run`: permette l'esecuzione di un Test esterno
- `set`: serve per settare temporaneamente una variabile con un parametro letto
- `verifyRows`: utilizzato nelle tabelle, serve per effettuare un confronto con delle liste

4.3.2.Sito Web, Documentazione, Materiale reperibile su Internet:

Il Sito Web è molto semplice da utilizzare. Sul sito vi è molta documentazione ed è spiegato tutto in modo molto chiaro. Su Internet non si trova molto materiale.

4.3.3.Licenza, Frequenza aggiornamenti, Linguaggi supportati, Portabilità:

Licenza di tipo Apache License, v2.0. Gli aggiornamenti sono abbastanza frequenti. I Linguaggi supportati sono i seguenti: Java, .Net, Python, Fantom, Ruby, Scala. Riguardo la portabilità, essendo scritto in java non ci sono problemi.

4.3.4.Facilità installazione & Integrazione Eclipse:

Uno dei punti di forza di Concordion è la facilità di installazione. Infatti su Eclipse basta importare i vari .jar per poter utilizzare Concordion. Tipicamente i Test vengono inseriti nel pacchetto "specs". Dal sito si scarica il pacchetto base, otterremo i vari .jar e i sorgenti. I vari file da includere sono i seguenti: (concordion-1.4.4.jar , junit-4.11.jar , hamcrest-core-1.3.jar , ognl-2.6.9.jar , xom-1.2.5.jar). Esiste un Plugin (Concordion Eclipse Plugin) che aiuta il programma nella scrittura dei Test, infatti

attraverso un CTRL + Space, comparirà un menu a tendina con tutte le funzioni invocabili presenti sul file Java che si occuperà dell'elaborazione.

4.3.5. Compatibilità con le Versioni Java:

Concordion è compatibile con le ultime versioni Java (Compreso la 8)

4.3.6. Facilità scrittura & formato Test:

Sicuramente i Test risultano difficili da scrivere per chi non ha un minimo di conoscenza informatica. I Risultati sono in formato XHTML, quindi facilmente esportabili. A differenza di FIT qui non si ha l'obbligo di utilizzare delle tabelle, quindi si ha sicuramente una maggiore esportabilità. Per la scrittura dei Test ci si può aiutare con degli editor WEB. Essendo scritto in XHTML, la fase di scrittura di un Software per effettuare il parsing dei risultati risulta molto più semplice. Concordion mette a disposizione un plugin sia per la scrittura dei Test attraverso l'utilizzo di tabelle Excel, ma anche la possibilità di esportarle nello stesso formato.

4.3.7. Estensibilità:

Dal punto di vista dell'estensibilità probabilmente Concordion è il migliore tra gli strumenti confrontati. Essendo Open Source è facilmente modificabile, si può poi estendere il set di comandi appartenenti al Namespace Concordion. Sul sito è spiegato in modo molto chiaro su come aggiungere comandi. Per aggiungere un comando bisogna implementare una specifica interfaccia ed implementare tutti i metodi necessari.

4.4. Confronto e scelta dello Strumento

Dopo un confronto approfondito dettagliato nel paragrafo precedente ed una valutazione in decimi, si è arrivati al risultato presente nella tabella 4.4.1.

	FIT	FITNESSE	CONCORDION
Sito Web, Documentazione sul Sito, Materiale su Internet	6	8	8
Licenza, Frequenza Aggiornamenti, Linguaggi Supportati, Portabilità	5	9	9
Facilità Installazione & Integrazione con Eclipse	9	7	9
Compatibilità con le Versioni Java	1	9	10
Facilità scrittura & formato Test	8	9	9
Estensibilità	5	8	9
Caratteristiche Generali	3	10	10
Valutazione "Extra"	4	8	10
TOTALE	41	68	74

Figura 4.4.1: Confronto Strumenti

Data la valutazione effettuata Concondion risulta lo strumento più completo presente sul mercato per la stesura di Test di accettazione.

Sicuramente FITNESSSE ha molte più caratteristiche che non sono presenti in Concondion, ma si tratta comunque di proprietà che non sono necessarie per problematiche ordinarie. In Concondion, a differenza di FITNESSSE, la scrittura delle Fixture risulta molto semplice ed il codice è molto più leggibile. Non si è obbligati ad utilizzare le tabelle, infatti per effettuare dei Test è anche possibile utilizzare i paragrafi, liste, ecc. Durante la scrittura dei Test, il programmatore può concentrarsi sul rendere leggibili i documenti, senza occuparsi di come dovranno essere elaborati. In FITNESSSE risulta impossibile una separazione netta tra codice e Test. In Concondion le funzioni che vengono chiamate sono esplicitate in modo molto chiaro, senza creare ambiguità che potrebbero indurre in errori il programmatore.

Tra i principali vantaggi nell'utilizzo di Concondion abbiamo la facilità di apprendimento e soprattutto la possibilità di esecuzione attraverso lo strumento JUnit, infatti uno dei problemi

principali di FITNESS è che, usando una WIKI, è necessario l'esecuzione di uno strumento esterno per la scrittura del codice e per la sua esecuzione.

5. Caso di studio: tuProlog

La Software selection degli strumenti per la scrittura di Test di accettazione ha portato alla selezione di Concordion. Il caso di studio che ne vuole mostrare l'efficacia sarà, come anticipato, la scrittura dei Test di accettazione per il sistema tuProlog. Nei due paragrafi successivi si riporta la progettazione e l'implementazione di questi Test.

5.1. Progettazione Test di accettazione

L'obiettivo è quello di riscrivere gli attuali Test di accettazione attraverso l'utilizzo del Framework Concordion. Partendo dagli attuali Test standard, è stato effettuato un merge con i Test scritti con il Framework FIT, facendone porting. Nel progetto tuProlog, le Fixtures sono presenti all'interno del package `Test/fit/fixtures`

Le classi Fixtures sono le seguenti:

- `AcceptanceFixture`: è la classe che si occupa di effettuare il parsing del codice HTML, questo in Concordion non è necessario e quindi si tratta di codice che non verrà riutilizzato.
- `EngineFixture`: classe che contiene i metodi principali per l'esecuzione dei Test. Data la struttura completamente differente di Concordion, il codice di questa classe non è stato riutilizzato, ma è servito solo per dare un'idea generale sul funzionamento di FIT con tuProlog.
- `EvaluationFixture`: metodi di comodo non necessari.
- `NullTerm`: classe di comodo utilizzata per risolvere problemi di compatibilità non presenti in Concordion
- `PrologActionFixture`: utilizzata per la gestione delle eccezioni, in Concordion verranno gestite in modo differente.

- SimpleGoalFixture: contiene un metodo che, dato un Goal, verifica che abbia successo o meno.

E' l'unico metodo che è stato riutilizzato anche in Concordion.

Riguardo le FIT Table, quest'ultime sono state riutilizzate, con le dovute modifiche. Infatti in Concordion, per la scrittura dei Test, viene utilizzato l'XHTML, con un namespace specifico. Si è dovuto quindi seguire delle regole per una corretta formattazione, ed in più è stato necessario specificare i metodi per l'elaborazione dei Test all'interno del codice. Nel nostro caso le classi Fixtures dovranno essere implementate in modo tale che:

- Si possa, data una query, verificare che vi sia presente una soluzione o meno
- Si possa, data una query e un risultato, verificare che quel determinato risultato sia presente nel set di soluzioni.
- Si possa, data in query, intercettare eventuali eccezioni

Questi sono i requisiti iniziali che già FIT permetteva di fare, ma con i suoi limiti legati all'età. Inoltre si vuole cercare di ottimizzare i tempi di esecuzione e di utilizzare tutti quegli strumenti come i tipi generici che con FIT non era possibile utilizzare.

5.1.1.Singleton

Uno dei problemi di Concordion è legato al vincolo nella scelta dei nomi dei file contenenti i Test e le classi Fixtures. Infatti si ha come limite che il file .html contenente i Test deve avere lo stesso nome del file .Java contenente tutti i metodi che serviranno per l'elaborazione dei Test. Un altro vincolo è legato al fatto che in ogni classe Fixtures bisogna specificare il codice dei metodi utili per l'elaborazione, anche se questi già presenti in altre classi. Questo causa una ripetizione di codice, il che crea facilmente confusione soprattutto una scarsa manutenibilità dei Test.

Per ovviare a questo inconveniente si è optato per l'utilizzo di un Singleton. Questo non risolve pienamente il problema, infatti in tutte le classi Fixtures bisognerà comunque specificare tutti i

metodi, ma per la loro invocazione se ne occuperà il Singleton, il che rende necessario più gestibile anche una eventuale reingegnerizzazione del codice.

Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punti di accesso globale a tale istanza. L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziatura diretta della classe. La classe fornisce inoltre un metodo di tipo `getter` statico che restituisce una istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico. Nel nostro caso il Singleton è stato chiamato `ConcordionSingleton` con il seguente codice:

```
public class ConcordionSingleton {  
  
    private static ConcordionSingleton singleton;  
    private String exceptionFounded = "";  
  
    private ConcordionSingleton() {  
    }  
  
    public static ConcordionSingleton getInstance() {  
        if (singleton == null) {  
            singleton = new ConcordionSingleton();  
        }  
  
        return singleton;  
    }  
}
```

Roberta Calegari 27/11/y 11:36

Commenta [1]: Formatta bene codice usa courier e piccolo

Nel Singleton sono presenti poi i vari metodi pubblici e privati utilizzati per l'elaborazione dei Test, i quali verranno spiegati approfonditamente nei prossimi paragrafi.

5.2. Implementazione Test di accettazione

Di seguito verranno esaminati nel dettaglio i metodi implementati per la scrittura dei Test.

In particolare i Test si possono classificare nelle seguenti macro categorie:

- Verifica di successo di un goal passato al motore prolog
- Verifica che la soluzione della query prolog sia analoga alla soluzione fornita
- Verifica che nella risoluzione di una query prolog si verifichi una eccezione

5.2.1. Scrittura classi Fixtures

5.2.1.1. Verifica di successo di un goal passato al motore prolog

Uno dei metodi base, ed anche uno dei più utilizzati, è stato ripreso dal codice FIT, inserendo la possibilità di passare come parametro una teoria. Il parametro verrà preso in considerazione solo se il valore passato è diverso da `null`.

```
public boolean success(String goal, String theory) throws Exception {  
  
    Prolog engine = new Prolog();  
    if (!theory.equalsIgnoreCase("null"))  
        engine.setTheory(new Theory(theory));  
    SolveInfo info = engine.solve(goal);  
    return info.isSuccess();  
}
```

Il metodo serve per verificare se, dato un goal ed eventualmente una teoria, esiste una soluzione.

Questo viene verificato utilizzando il metodo `isSuccess()` della `SolveInfo`. In FIT non era

possibile svolgere un'operazione del genere con una sola chiamata ad un metodo, infatti erano necessarie due operazioni, una per il settaggio della teoria e uno per la verifica.

5.2.1.2. Verifica che la soluzione della query prolog sia analoga alla soluzione fornita

Un altro metodo necessario per l'esecuzione dei Test è quello che verifica se dato un goal, eventualmente una teoria, una variabile e una soluzione, che quella determinata soluzione sia presente all'interno del set delle soluzioni di quello specifico goal. In questo caso sono stati vari i problemi emersi, principalmente due:

In tuProlog i risultati spesso non vengono mostrati correttamente, infatti vengono aggiunti dei numeri casualmente, i quali rendono impossibile effettuare un confronto. Questo è uno dei problemi che è rimasto senza soluzione in FIT. (Figura 5.2.2.1)

check	binding	<code>[f(a,_).f(_b)]</code> <i>expected</i>
		<code>[f(a_1488600949),f(_731662293,b)]</code> <i>actual</i>

Figura 5.2.2.1

L'altro problema emerso è stato che in alcune query, il set di soluzioni è infinito, e quindi si rischia di finire in un loop senza via di uscita.

Per la risoluzione del primo problema si è deciso di utilizzare due metodi che si occupano di effettuare un replace, ovvero di eliminare tutti quei numeri e caratteri aggiunti dopo il carattere '_' . Questo replace è anche disattivabile, dato che in alcuni casi quei numeri dopo l'underscore potrebbero far parte della soluzione. Il codice dei metodi è il seguente:

```
private String replaceForVariable(String query, char car) {  
  
    String result = "";
```

```

        for (int i = 0; i < query.length(); i++) {

            if (query.charAt(i) == car)
                return result;
            result += (query.charAt(i) + "");

        }

        return result;
    }

    private String replaceUnderscore(String query) {

        String result = "";
        boolean trovato = false;
        for (int i = 0; i < query.length(); i++) {

            if (query.charAt(i) == ',' || query.charAt(i) == ')' ||
                query.charAt(i) == ']')
                trovato = false;
            if (!trovato)
                result += (query.charAt(i) + "");
            if (query.charAt(i) == '_')
                trovato = true;

        }

        return result;
    }
}

```

Il primo metodo è utilizzato per effettuare la sostituzione con un determinato carattere passato come parametro, il secondo si occupa di risolvere il problema dell'underscore. Dato che si potrebbe non volere il replace in alcuni casi, sono stati creati dei “duplicati” dei metodi con il replace disattivato.

In questo caso si è deciso di creare ulteriori metodi, invece che passare il parametro direttamente nel codice HTML, perché altrimenti si rischiava di aggiungere un numero elevato di colonne di scarsa utilità. L'invocazione di un metodo senza replace è un'operazione rara, ed è a carico del programmatore. Per la risoluzione delle query con un numero infinito di soluzioni, è stato introdotto un metodo in cui è possibile passare un parametro che indica il limite. Verrà effettuata la verifica che quella determinata soluzione sia presente all'interno delle prime N soluzioni, con N passato come parametro. (In questo caso direttamente nel codice XHTML). Quindi il codice del metodo che si occupa di verificare che una determinata soluzione appartenga al set di soluzioni di una query è il seguente:

```
private boolean successAndResultsWithLimitVerifyReplace(String goal,
    String theory, String variable, String solution, boolean
    replace, int maxSolutions) throws Exception {

    Prolog engine = new Prolog();
    SolveInfo info = null;
    List<String> results = new ArrayList<String>();

    if (!theory.equalsIgnoreCase("null"))
        engine.setTheory(new Theory(theory));
    info = engine.solve(goal);
    while (info.isSuccess() && maxSolutions != 0) {

        for (Var var : info.getBindingVars()) {
            if ((var.toString()).startsWith(variable)) {

                variable = (replace == true ?
replaceForVariable(
                    var.toString(), ' ') :
var.toString());

                Term t = info.getVarValue(variable);
                results.add(replace == true ?
replaceUnderscore(t
```

```

        .toString()) : t.toString());

    }

}

if (replace)
    variable = replaceForVariable(variable, '_');
Term t = info.getVarValue(variable);
results.add(replace == true ?
replaceUnderscore(t.toString()) : t
.toString());

if (engine.hasOpenAlternatives()) {
    info = engine.solveNext();
} else {
    break;
}

maxSolutions--;
}

return results.contains(solution);
}

```

In questo caso è stata utilizzata una lista per contenere il set di soluzioni. Al termine del metodo viene effettuata la verifica che la soluzione data sia presente all'interno della lista. Nel codice viene sempre verificato che il replace sia effettivamente attivato, inoltre durante il ciclo, vi è un conteggio nel numero delle soluzioni e nel caso in cui si superi il valore MAX passato come parametro, si uscirà immediatamente dal ciclo e si effettuerà il confronto.

5.2.1.3. Verifica che nella risoluzione di una query prolog si verifichi una eccezione

L'altro metodo necessario per l'esecuzione dei Test è quello che verifica se si è verificata una eccezione, ed eventualmente la segnali. In FIT la gestione delle eccezioni non funzionava molto bene, infatti veniva effettuato un confronto se l'errore verificatosi combaciava con quello passato come parametro. Questa è un'operazione che non sempre possiamo fare, infatti di solito conosciamo solo in generale l'errore, inoltre spesso con le nuove versioni del Software, vengono effettuate delle modifiche nell'output delle eccezioni, il che renderebbe i Test obsoleti. (Figura 5.2.2.2)

alice.tuprolog.PrologActionFixture		
enter	query	atom_chars(X, Y).
check	hasSolution	false
check	exception	instantiation_error

Figura 5.2.2.2

In questo caso si è deciso che attraverso i Test si specifichi se quella determinata operazione possa dare una eccezione o meno, quindi non si specifica il possibile errore, ma in caso di eccezione verrà mostrato in output da linea di comando (In questo caso di Eclipse), l'errore verificatosi. Durante la progettazione sono emerse alcune problematiche, infatti, tuProlog non ha una gestione molto semplice delle eccezioni. L'entità che cambia stato durante l'esecuzione è il motore inferenziale Engine e il valore dello stato è definito da un insieme di costanti definite nella classe EngineManager. Il motore dispone di un meccanismo per scambiare messaggi. Il modello utilizzato si basa su tre elementi: evento, gestore e ascoltatore. Attualmente sono definiti eventi e relativi ascoltatori per notificare informazioni di solution, output, spy, e warning. La classe Prolog contiene la logica per gestire eventi ed ascoltatori. Ogni componente dell'applicazione che voglia essere sensibile ad un tipo di evento deve implementare la relativa interfaccia e richiedere alla classe Prolog di essere inserito nella coda degli ascoltatori relativi. A tale scopo, per ciascun evento,

vengono definiti un ArrayList contenitore di ascoltatori, i metodi accessori per aggiungere, rimuovere o restituire la lista degli ascoltatori, una variabile di stato con metodi di “set” e “get” per indicare se il motore debba notificare quel tipo di evento e un metodo per passare alla classe Prolog il messaggio da inviare a tutti gli ascoltatori di un certo tipo. Per replicare questo schema anche per gestire le eccezioni è stato definito nel package `alice.tuprolog.event` la classe `ExceptionEvent`, che estende `EventObject` e contiene un campo String per contenere il messaggio d’errore, e la classe `ExceptionListener`, che estende `EventListener` e contiene il metodo `public abstract void onException(ExceptionEvent)`. A questo punto è possibile aggiungere alla classe Prolog gli elementi necessari per gestire gli ascoltatori e gli eventi riguardo delle eccezioni al pari degli eventi già trattati.

Per poter notificare agli `ExceptionListener` i messaggi relativi alle eccezioni occorre farle giungere dal punto in cui vengono catturate (classe `StateGoalEvaluation`) fino al gestore (classe Prolog).

Sfruttando il riferimento all’istanza Engine, passato come parametro nel metodo `doJob()` della classe `StateGoalEvaluation`, si può accedere all’istanza `EngineManager`. Quest’ultima si occupa già di notificare l’istanza Prolog dei messaggi di spy e di warning. È sufficiente, quindi, aggiungere il metodo `void exception(String)` per passare un messaggio relativo ad un’eccezione.

Vi sono quindi varie possibilità per la gestione delle eccezioni:

- Aggiunta del listener nei Test riutilizzando i componenti e i metodi già esistenti in `tuProlog`

Questa è stata la soluzione adottata come definitiva. Nel codice dei Test è stato aggiunto un listener, implementando i metodi necessari, e durante l’esecuzione del metodo che si occupa di verificare se vi è stata una eccezione o meno, viene controllato che il listener abbia intercettato delle eccezioni.

```
private String exceptionFounded = "";
```

```

private boolean exFounded = false; // variable used to found an
exception
private ExceptionListener ex = new ExceptionListener() {

    @Override
    public void onException(ExceptionEvent e) {
        exFounded = true;
        exceptionFounded = e.getMsg();
    }
};

```

La stringa `exceptionFounded` serve per salvare l'eventuale eccezione verificatasi durante l'esecuzione del metodo. E' stato creato un oggetto di tipo `ExceptionListener`, nel quale è stato implementato il metodo vero e proprio (`onException`) che si occupa di intercettare le eventuali eccezioni. Sicuramente non è la migliore delle soluzioni, ma si tratta di una soluzione temporanea, in attesa che venga aggiornato `tuProlog`.

Il codice del metodo per individuare le eccezioni e mostrarle in Output è il seguente:

```

public String successWithExceptionAndText(String goal, String theory)
    throws PrologException {

    Prolog engine = null;
    @SuppressWarnings("unused")
    SolveInfo info = null;
    exFounded = false;
    engine = new Prolog();
    if (!theory.equalsIgnoreCase("null"))
        engine.setTheory(new Theory(theory));
    engine.addExceptionListener(ex);
    info = engine.solve(goal);
    if (exFounded)
        return exceptionFounded;
    else
        return "No Errors!";
}

```

```
}
```

Viene verificato che ci sia stata una eccezione, ed eventualmente viene stampato l'errore

- re-ingegnerizzare tuProlog pesante attraverso la creazione di un componente a parte per catturare le eccezioni e dare diversa visibilità al motore all'esterno.

In questo caso si tratta solo di un'idea per la risoluzione del problema, senza andare ad effettuare delle prove e a scrivere codice.

- re-ingegnerizzare tuProlog leggera aggiungendo alla solveInfo un metodo di tipo getter per recuperare la lista delle eccezioni che si sono verificate durante la risoluzione della query.

Sicuramente la prima cosa che viene in mente, durante la scrittura del codice, per intercettare le eccezioni, è di fare una sorta di `getErrors()` alla solveInfo, questo però non è possibile. Per renderlo possibile è necessario inserire il seguente codice all'interno del file Prolog.java

```
protected void notifyException(ExceptionEvent e) {  
    for (ExceptionListener el:exceptionListeners) {  
        el.onException(e);  
    }  
  
    errors.add(e.getMsg());  
}  
  
private List<String> errors = new ArrayList<String>();  
  
public List<String> getErrors() {  
  
    return errors;  
}
```

Sostanzialmente viene inserito il listener, e le eventuali eccezioni inserite in una lista. Poi attraverso una `getErrors()` è possibile recuperare tutti gli errori. In questo caso il codice del metodo sarebbe il seguente:

```
public boolean successWithException(String goal, String theory)
    throws PrologException {

    Prolog engine = null;
    @SuppressWarnings("unused")
    SolveInfo info = null;
    exFounded = false;
    engine = new Prolog();
    if (!theory.equalsIgnoreCase("null"))
        engine.setTheory(new Theory(theory));
    engine.addExceptionListener(ex);
    info = engine.solve(goal);
    System.out.println(engine.getErrors());
    return exFounded;

}
```

Questa si tratta comunque di una modifica non implementata, infatti è stata fatta solo per scopo di Test, in attesa di una nuova versione di tuProlog.

5.2.2. Scrittura Test XHTML

La seconda fase della scrittura dei Test Concondion è quella di scrivere la tabella (o qualsiasi altra struttura, dato che in Concondion non vi è il limite sulle strutture utilizzabili) che conterranno i dati veri e propri da collaudare. A differenza di FIT, si tratta di un file XHTML, andrà quindi specificato il Namespace e seguite determinate regole. Nel nostro caso abbiamo utilizzato come strutture solo le tabelle. Nel tag `<table>` bisogna specificare attraverso il comando `execute`, il metodo che dovrà essere invocato, la variabile in cui verrà settato il risultato e i parametri, i cui nomi dovranno combaciare con quelli presenti nel `<th>`. Nel tag `<th>` attraverso il comando `set` viene effettuata la lettura del parametro presente all'interno del tag e settato all'interno di una variabile (che verrà poi

passata nel momento in cui verrà invocato il metodo. L'altro comando fondamentale è l'`assertTrue`, che si occupa di verificare che il risultato del metodo invocato sia vero. Se i risultati combaciano, la casella verrà illuminata di verde, altrimenti di sorto. Oltre l'`assertTrue`, esiste l'analogo `assertFalse` e l'`assertEquals` che permette di effettuare un confronto, tenendo conto anche della tipizzazione dei dati.

Un esempio di codice XHTML, per verificare che una determinata query abbia soluzione è il seguente:

```
<table concordion:execute="#result = success(#goal,#theory)">
  <tr><th concordion:set="#goal">Goal</th>
    <th concordion:set="#theory">Theory</th>
      <th concordion:assertTrue="#result">success(String
goal,String theory)</th></tr>
  <tr>
    <td>'=\='(0, 1).</td>
    <td>null</td>
    <td>true</td>
  </tr>
```

Nella figura 5.2.3.1 è possibile vedere l'output del file, prima che venga elaborato.

1.1 Example tests

Let's start with some simple tests verifying success or failure of single goals.

Goal	Theory	success(String goal,String theory)
'\='(0, 1).	null	true
'<'(0, 1).	null	true
'=<'(0, 1).	null	true
'\='(1.0, 1).	null	true
'>='(1.0, 1).	null	true
'=<'(1.0, 1).	null	true
'\='(3 * 2, 7 - 1).	null	true
'>='(3 * 2, 7 - 1).	null	true
'=<'(3 * 2, 7 - 1).	null	true

Figura 5.2.3.1

In Figura 5.2.3.2 una tabella utilizzata per verificare che vi siano delle eccezioni. Nella quarta colonna compariranno le eventuali eccezioni, per l'esecuzione di un metodo e la visualizzazione in output del risultato è stato utilizzato il comando `ext:embed`.

Test With Exception

Goal	Theory	success(String goal)	Type Of Error
'\='(X, 5).	null	true	
'\='(X, 5).	null	true	
'<'(X, 5).	null	true	
'>'(X, 5).	null	true	
'>='(X, 5).	null	true	
'=<'(X, 5).	null	true	

Figura 5.2.3.2

Un altro limite di Concordion riguarda la possibilità di esecuzione di un solo file Testing per volta. Questo è un grosso limite, infatti tipicamente dopo una modifica al codice tuProlog, si vuole

effettuare una verifica di tutti i Test, per individuare eventuali nuovi errori emersi. Per risolvere questo problema, è stato introdotto un file apposito, General. html e il corrispettivo General.Java. Per seguire tutti i Test basta eseguire questo General, il cui codice Java è vuoto, ma il codice html è il seguente

```
<ul>
  <li>
    <a concordion:run="concordion" href="AllSolutions.html">
      AllSolutions
    </a>
  </li>
  <li>
    <a concordion:run="concordion" href="ArithmeticComparison.html">
      Arithmetic Comparison
    </a>
  </li>
  <li>
    <a concordion:run="concordion" href="ArithmeticEvaluation.html">
      ArithmeticEvaluation
    </a>
  </li>
  <li>
    <a concordion:run="concordion" href="ListManagement.html">
      List Management
    </a>
  </li>
  <li>
    <a concordion:run="concordion" href="TermComparison.html">
      Term Comparison
    </a>
  </li>
  <li>
    <a concordion:run="concordion" href="TermCreationandDecomposition.html">
      Term Creation andDecomposition
    </a>
  </li>
</ul>
```

```

<li>
  <a concordion:run="concordion" href="TypeTesting.html">
    Type Testing
  </a>
</li>
<li>
  <a concordion:run="concordion" href="TermUnification.html">
    Term Unification
  </a>
</li>
</ul>

```

Il file in output conterrà i link ai vari file Testing eseguiti. In Figura 5.3.2.3 il General prima dell’elaborazione.

List of Concordion Test

List:

- [AllSolutions](#)
- [Arithmetic Comparison](#)
- [ArithmeticEvaluation](#)
- [List Management](#)
- [Term Comparison](#)
- [Term Creation andDecomposition](#)
- [Type Testing](#)
- [Term Unification](#)

Figura 5.2.3.3

5.3. Collaudo

Per l'esecuzione dei Test bisogna eseguire, attraverso JUnit, i file .Java (Figura 6.1.1). Una volta elaborati i file, nella console comparirà il path dei file in output e gli eventuali errori verificatosi. (Figura 6.1.2 e 6.1.3)

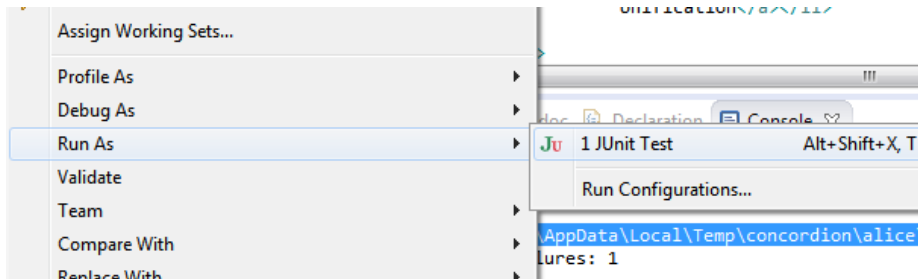


Figura 6.1.1

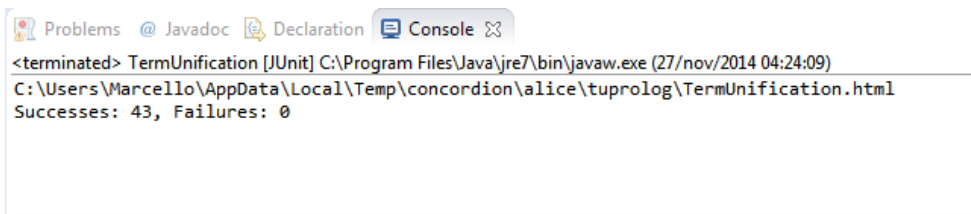


Figura 6.1.2

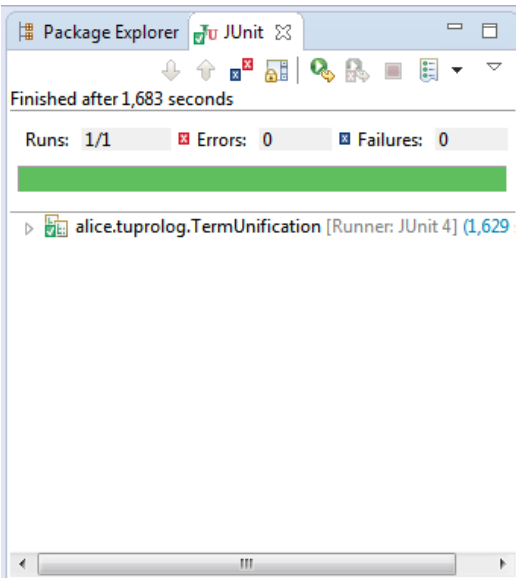


Figura 6.1.3

Un esempio di file in output prodotto è il seguente (Figura 6.1.4)

1.1 Example tests

Let's start with some simple tests verifying success of failure of single goals.

Goal	Theory	success(String goal,String theory)
'=(1, 1).	null	true
'=(_, _).	null	true
'=(X, 1).	null	true
'=(X, Y).	null	true
'=(X=Y, X=abc).	null	true
'=(f(X,def), f(def,Y)).	null	true

Figura 6.1.4

Il file General una volta elaborato (Figura 6.1.5)

List of Concordion Test

List:

- [AllSolutions](#)
- [Arithmetic Comparison](#)
- [ArithmeticEvaluation](#)
- [List Management](#)
- [Term Comparison](#)
- [Term Creation andDecomposition](#)
- [Type Testing](#)
- [Term Unification](#)

Figura 6.1.5

6. Conclusioni

Il primo obiettivo della tesi era di quello di effettuare un confronto tra una serie di strumenti utilizzati per la scrittura dei Test di accettazione. In questo caso l'obiettivo è stato raggiunto con la scelta del Framework Conordion. Il secondo obiettivo era quello di validare lo strumento selezionato in un caso reale di programmazione multi paradigma, applicativo tuProlog multi-paradigma e multi-piattaforma dalla nascita. Relativamente al caso di studio sono stati riscritti i Test di accettazione attraverso un'unione tra gli attuali Test FIT presenti, e quelli standard. Alcuni problemi che erano prima presenti con i vecchi Test sono stati risolti, precisamente:

- Risoluzione del bug dell'underscore, in questo caso si tratta di una risoluzione momentanea. Per evitare di fare in futuro molte modifiche, sono stati creati dei metodi appositi in cui i caratteri speciali non vengono sostituiti.
- La gestione delle eccezioni, infatti in FIT non sempre venivano intercettate. In questo caso vengono sia intercettate, che mostrato l'errore completo.

Durante la progettazione per gestire le eccezioni, sono emerse alcune problematiche, legate alla loro gestione nel Framework tuProlog. La soluzione per cui si è optato è stata quella di aggiungere un listener nel Test riutilizzando i componenti e i metodi già esistenti in tuProlog. Nel codice dei Test è stato aggiunto un listener, implementando i metodi necessari, e durante l'esecuzione del metodo che si occupa di verificare se vi è stata una eccezione o meno, viene controllato che il listener abbia intercettato delle eccezioni.

A livello di studio generale della soluzione si sono esplorate due alternative ad impatto maggiore:

- Reingegnerizzazione pesante del motore tuProlog attraverso la creazione di un componente a parte per catturare le eccezioni e dare diversa visibilità al motore all'esterno. In questo caso si

tratta solo di un'idea per la risoluzione del problema, senza andare ad effettuare delle prove e a scrivere codice.

- Reingegnerizzazione minimale del motore tuProlog aggiungendo alla solveInfo un metodo di tipo getter (`getErrors()`) per recuperare la lista delle eccezioni che si sono verificate durante la risoluzione della query.

Tra i punti aperti, vi sono dei test che non restituiscono il valore atteso. In questo caso bisognerà investigare la causa del problema, probabilmente legata a difetti del motore.