UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria

Corso di Studi in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

Stato dell'arte
e valutazione di strumenti open source
per il test di web application

Anno Accademico 2011/2012

Candidata:

Liana Allocca

matr. N46/295

Ai miei genitori,
a Mario,
a me stessa

Indice

Commento [P1]: In seduta di laurea ci tengono molto al che la tesi non sia molto oltre le 25 pagine. Cerca di ridurre di qualche pagina, magati snellendo la parte introduttiva e riducendo il margine sinistro (che mi sembra un po' eccessivo, anche tenuto conto della rilegatura)

Introduzione

Negli ultimi anni siamo stati spettatori della crescita esponenziale di Internet e nello stesso tempo dello sviluppo della tecnologia e della diffusione di nuovi mezzi di comunicazione. Internet è entrata nelle case di tutti, proprio come avvenne in passato con la radio e la televisione, e questo ha contribuito alla sua sconfinata espansione. Inizialmente usata per scopi militari attraverso il collegamento di un numero limitato di calcolatori, attualmente è la risorsa virtuale più grande che si potesse immaginare di avere a disposizione. Con la nascita del World Wide Web si ha assistito ad un progresso tecnologico che ha superato di gran lunga le aspettative dei suoi padri. Internet, attualmente, è usata dall'utente "on demand" (su richiesta): il cliente ottiene il servizio che vuole, quando vuole, spaziando dallo studio, al lavoro, allo svago, servendosi dei più noti motori di ricerca e delle innumerevoli applicazioni web disponibili.

Le *applicazioni web* sono sicuramente la parte di Internet che l'utente della rete "conosce meglio", merito anche della loro interfaccia sempre più semplice ed intuitiva (user-friendly). Attraverso le applicazioni web, il cliente può registrarsi ad un sito d'interesse, ricevere proposte di lavoro, inviare e-mail, condividere documenti personali, chattare con gli amici, effettuare chiamate, fare acquisti on-line, e tanto altro… tutto in modo facile e soprattutto gratuito. Questo aspetto ha notevolmente favorito la sua diffusione globale e la "curiosità" di molti, che li ha

spinti ad approfondire, cercare, andare oltre quell'interfaccia… Non serve essere programmatori esperti, ma semplicemente disponendo degli strumenti adatti si potrebbe creare una pagina web, una piccola applicazione, un sito d'interesse… Tali strumenti, fondamentali dunque per lo sviluppo e l'innovazione, sono persino reperibili in rete! E' proprio questa l'idea vincente del software *open source*: gli autori lo rendono disponibile affinché tutti gli utenti della rete possano usufruirne. Per i normali utenti, l'espressione "open source" si traduce sostanzialmente in "gratuito". Per i programmatori, invece, questa espressione apre tutta un'altra visione, molto più ampia e complessa. Se provvisti delle apposite licenze d'uso, questi possono apportarvi modifiche e miglioramenti, allo scopo di tenere il prodotto sempre aggiornato, aumentare la sua flessibilità e collaborare al lavoro di gruppo e alla condivisione della conoscenza.

Esistono innumerevoli categorie di *strumenti open source* reperibili, ma in questa trattazione ci si interesserà di quelli per lo *sviluppo web*, ponendo particolare attenzione agli strumenti che automatizzano il *processo di test* di tali applicazioni. Se svolta in modo manuale l'attività di test è molto impegnativa e potrebbe risultare interminabile. Per questo motivo attualmente la si ottimizza mediante strumenti automatici. Strumenti che testano applicazioni web basandosi sull'interazione con l'utente spesso implementano l'approccio "Capture and Replay".

Allo *stato dell'arte* si individuano strumenti come HtmlUnit e Selenium, rispettivamente l'uno appartenente alla famiglia XUnit e l'altro implementato come estensione del noto browser Mozilla Firefox. Si affrontano nei loro funzionamenti basilari e attraverso un'analisi comparativa, basandosi su un particolare caso di studio.

Infine sono illustrati due ulteriori strumenti di confronto, basandosi su aspetti più o meno similari ai due strumenti principali.

Capitolo 1

L'automazione nel test di un software

In questo primo capitolo si richiama brevemente il concetto di test di un software, ricordandone le definizioni e le varie tipologie. Successivamente si affronta in modo più approfondito il tema dell'automazione nell'ambito del test.

1.1 Definizioni: Errore, Difetto, Malfunzionamento

Nella sua accezione più ampia, il testing si effettua per valutare la qualità di un'applicazione, scoprendo i problemi che affliggono il software sotto esame.

Ma la parola "problema", in questo contesto, risulta essere troppo generica. Quindi, è necessario particolarizzarla per evitare confusione nella comprensione dell'argomento. Si ricordano le seguenti definizioni:

- *Errore* (error): è l'incomprensione umana in termini di requisiti software o nell'uso di strumenti;
- *Difetto* (fault o bug): si definisce tale la manifestazione dell'errore nel software;
- *Malfunzionamento* (failure): è l'incapacità del software di comportarsi secondo le aspettative o le specifiche; può essere osservato solo attraverso l'esecuzione.

1.2 Aspetti e strategie di testing

Nello sviluppo di un'applicazione software il testing, come già accennato, è un'attività tanto complessa quanto imprescindibile. In genere si compone di più fasi (Figura 1): si inizia con la progettazione dei casi di test, scegliendo particolari parametri di input del sistema e stimando quali saranno i relativi output nel caso in cui il sistema operi secondo le sue specifiche; infine si termina con la valutazione dei risultati ottenuti. Il test avrà avuto successo se il suo obiettivo sarà stato raggiunto: rivelare la presenza di malfunzionamenti nei sistemi. Ma, comunque, nulla garantirà dalla loro assenza.

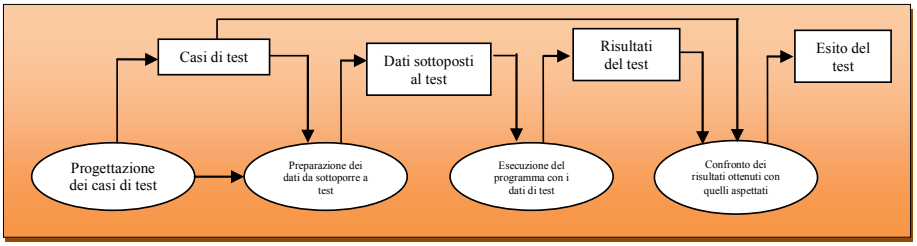


Figura 1: Processo di testing

Non esiste un momento temporale preciso nel progetto di un'applicazione in cui ha inizio il test: l'approccio migliore suggerirebbe di testare il software ad ogni fase dello sviluppo, per collaudare passo dopo passo il comportamento del sistema. Esistono varie strategie di testing: ciascuna di esse agisce su un particolare aspetto. Tra gli aspetti di un sistema più consistenti ci sono:

- *L'analisi delle singole componenti e della loro integrazione*: si realizza mediante test di unità e test di interfaccia;
- *La conoscenza delle funzionalità interne*: si realizza mediante test "white box" oppure "black box";
- *La valutazione delle prestazioni*: mediante test prestazionale e test di stress;

In generale, in un sistema molto complesso sarà difficile (se non impossibile) testare tutte le sue componenti contemporaneamente. Se anche fosse fattibile, probabilmente si manifesterebbero parecchie difficoltà nell'individuare della

sorgente dell'errore. Per questo nasce il test più semplice, il *test delle unità,* che prevede l'analisi di una singola unità del programma per volta. Un'unità può rappresentare un isolato programma, una funzione, una procedura, o anche un modulo. Quando il sistema è composto da diversi oggetti interagenti, si accede alle loro funzionalità attraverso la definizione di un'interfaccia. Effettuare un test su tale interfaccia è fondamentale per verificare il corretto funzionamento dell'intero sistema: gli errori riscontrabili nell'uso di un'interfaccia potrebbero essere molteplici poiché legati alla collaborazione fra elementi diversi… Dunque, un caso particolare di test delle unità è il *test delle interfacce*.

Per *test strutturale o "white box"* si intendono quei casi in cui è nota la struttura interna del programma (moduli, funzioni, procedure…) e quindi si cerca di scoprire gli errori logici commessi dal programmatore.

Nel *test funzionale o "black box"* non si ha la conoscenza di come il sistema sia strutturato; esso è visto come una scatola nera. Il test verrà progettato avendo presenti soltanto le specifiche del sistema.

Il *test prestazionale* si occupa di verificare che il sistema possa supportare una ingente quantità di carico di lavoro. In questo tipo di test, si tende ad aumentare via via il carico, fino a che le prestazioni del sistema diventano intollerabili.

Un'ulteriore strategia di test è il *test di regressione*, che ha senso effettuare nel momento in cui si apportano modifiche al progetto iniziale. Il termine "regressione" spiega bene il concetto base di questo approccio: verificare se, a seguito di una qualche modifica nel programma, la qualità del software sia regredita, cioè se siano stati introdotti dei difetti non presenti nella versione precedente alla modifica. La strategia consiste nel riapplicare al software modificato i test progettati per la sua versione originale e confrontare i risultati.

1.3 L'automazione dei test

E' frutto dell'automazione tutto ciò che è possibile gestire mediante dei meccanismi che riducono o addirittura sostituiscono l'intervento dell'uomo.

Sebbene spesso si faccia uso dell'automazione inconsapevolmente, sono davvero innumerevoli i vantaggi per la società moderna discesi da questa tecnologia, fra tutti l'ottimizzazione dei tempi e della qualità di esecuzione delle operazioni. Grazie allo sviluppo tecnologico, l'automazione è stata applicata anche al processo di test del software. *L'automazione del test è l'insieme delle operazioni svolte da un apposito strumento software per controllare il corretto funzionamento di un altro software o applicazione. Automatizzare un test è fondamentale per compiere un lavoro più rapido ed efficace.* Si pensi soltanto a quanto si riduce il rischio di incorrere in errori non rilevati dall'uomo! E' straordinario, no?

Essenzialmente, sono tre le aree in cui l'automazione interviene:

1. *Definizione dei casi di test*;
2. *Esecuzione del test*;
3. *Valutazione dell'esito del test.*

1.3.1 Definizione dei casi di test

Un test non si improvvisa, ma va studiato e preparato. Affinché sia efficace, deve essere realizzato tenendo presenti le caratteristiche del programma da esaminare. La definizione dei casi di test è perciò mirata a progettare un particolare contesto di esecuzione, selezionando un insieme di dati di input da fornire al programma. Definire i casi di test manualmente è sicuramente un'attività lunga e complessa.

L'uso dell'automazione già a partire da questa fase da luogo a molti vantaggi, ma comunque bisogna porre attenzione ai casi di test generati, che potrebbero risultare poco efficaci rispetto al contesto e pertanto inutili. Allora è necessario che essi siano definiti in modo meticoloso e ragionato, e supportati da strumenti software particolarmente avanzati. Tuttavia, per sperare in un esito positivo del test, non è necessario disporre di una gran quantità di casi su cui lavorare. E' più conveniente averne in numero inferiore, ed eventualmente combinarli per ottenerne di nuovi. Così si riduce sensibilmente il volume dei casi da valutare e quindi il tempo impiegato nel test.

1.3.2 Esecuzione del test

L'esecuzione di un test è governata da una struttura gerarchica chiamata *scaffolding*, che si compone di tre livelli (Figura 2):

1. Driver;
2. Unità oggetto del test;
3. Stub.

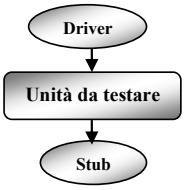


Figura 2: Scaffolnding

Un *driver* (o modulo guida) simula la parte di programma che invoca l'*unità da testare*, in particolare preparandone l'ambiente di esecuzione (per esempio, crea ed inizializza variabili globali, apre file…). Lo *stub* (o modulo fittizio) simula la parte di programma invocata dall'unità: verifica l'ambiente di esecuzione preparato dal driver e l'accettabilità dei parametri passati e restituisce i risultati. Infine, il driver verifica i risultati ottenuti e li memorizza.

Per l'esecuzione automatica dei test si utilizzano appositi *framework* (strutture di supporto al software complete di librerie e ambiente di sviluppo integrato) che simulano il comportamento descritto. Oggi esistono innumerevoli framework dediti all'attività di test (definizione test-cases, esecuzione test, valutazione esito). Si valuterà in particolar modo un framework della famiglia XUnit, che nasce proprio come supporto open source per il test di applicazioni web: *HtmlUnit*. Oltre a questo, esistono altri strumenti di natura diversa ma comunque efficienti: uno di essi è *Selenium*, che è implementato come estensione del browser Mozilla Firefox. I due strumenti qui citati verranno ripresi nel corso della trattazione.

1.3.3 Valutazione dell'esito del test

Per poter valutare i risultati del test è necessario conoscere il comportamento atteso del programma e confrontarlo con quello osservato. E' questo il concetto su cui si basa l'*oracolo*, che conosce il comportamento atteso per ogni caso di test. Utilizzando uno specifico set di dati di ingresso al sistema, l'oracolo identifica un determinato set di dati di uscita, che rappresentano il comportamento atteso.

L'esito di un test può essere valutato secondo due diverse metriche:

1) *L'efficacia di un test,* valutata come copertura dei casi di test;

2) *L'utilità di un test all'interno dell'intero progetto*, valutata come percentuale di copertura del codice (code coverage).

Commento [P2]: Quesre due definizioni non sono corrette

Il primo caso è presto spiegato: se durante il processo di test si riescono a "coprire" tutti i test case progettati, allora si è raggiunta una buona copertura per questi. Ciò vuoldire che il software è stato testato in modo corretto e che si è scoperto un numero elevato di malfunzionamenti. Nel secondo caso, invece, si cerca di stabilire la percentuale di linee di codice del progetto che sono state eseguite dai test. In un test solo il 100% di code coverage garantisce di aver individuato tutti i malfunzionamenti. Ma questo è un evento quantomeno improbabile.

1.4 Come migliorare il test: Approcci Capture and Replay

La valutazione dell'esito del test è la fase più critica dell'intero processo di test. In termini di copertura dei casi di test, solo se l'esito è positivo, comportamento atteso e osservato coincidono; in termini di code coverage, solitamente si raggiungono risultati accettabili anche con percentuali inferiori (70-80%). Pertanto bisogna trovare un modo per migliorare le prestazioni dell'intera fase di test, riducendo il rischio di avere alla fine comportamenti inaspettati. Come si potrebbe fare?

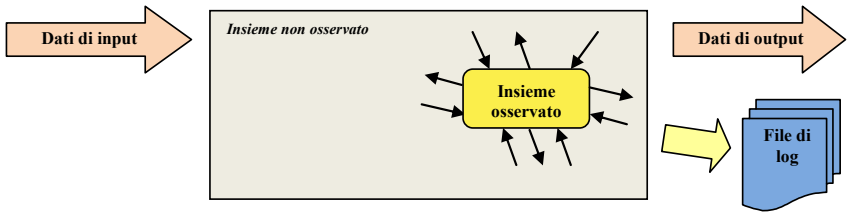
Esiste un approccio noto come *"Capture and Replay"* (cattura e replicazione) sul quale si basano molti strumenti di test all'avanguardia. Questa tecnica permette di migliorare l'intero processo, partendo dall'osservazione diretta dell'esecuzione dei programmi.

1.4.1 La fase di Capture

Monitorando il comportamento di un'applicazione durante l'esecuzione è possibile memorizzarlo ed utilizzare le informazioni raccolte come casi di test: questa è la *fase di Capture*. E' bene sapere però che la cattura dell'intera esecuzione è generalmente un evento irrealizzabile, poiché richiede di aver traccia di un volume di dati troppo consistente. Per questo motivo, si utilizza un sottoinsieme di dati

d'interesse, detto *insieme osservato*, che comprende solo i dati che potrebbero intaccare il testing. Si definiscono i confini dell'insieme osservato e si applicano modifiche al codice per riuscire a catturare le interazioni tra insieme osservato e *insieme non osservato*: tali modifiche vengono chiamate *sonde* (probes). La tecnica non solo osserva il comportamento dinamico del sistema, ma attua anche variazioni all'interno del codice per poterlo monitorare. Difatti, proprio le sonde permettono la generazione dei dati da memorizzare nei *file di log*. I dati presenti in tali file sono dunque corrispondenti non all'intera osservazione, ma solo alle azioni del sistema che "attraversano" i confini. Interazioni tipiche sono: chiamate di metodi, accesso ai campi ed eccezioni. La fase di Capture descritta è spesso detta "cattura selettiva" (Figura 3).

Figura 3: La fase di Capture



1.4.2 La fase di Replay

La *fase di Replay* segue quella di Capture e prevede l'identificazione di tutte le interazioni e la successiva replicazione degli eventi generati. Il file di log, creato in fase di Capture, viene letto in maniera sequenziale: ciascuna istanza di cui è composto viene singolarmente analizzata. Successivamente, servendosi dei principi della gerarchia scaffolding, si esegue il test *replicando l'evento registrato* e riproponendo in tal modo il comportamento del sistema (Figura 4).

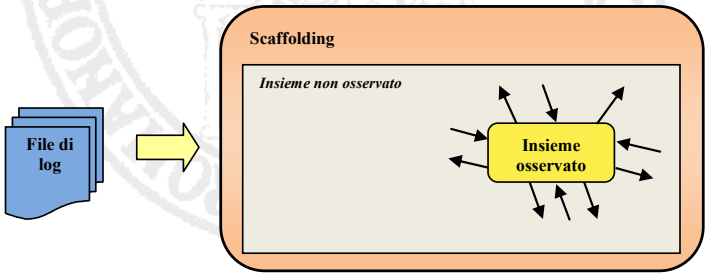


Figura 4: La fase di Replay

Infine, si effettua il confronto: se la simulazione produce un risultato identico alla prima istanza registrata del file di log, allora significa che l'esecuzione si è replicata identicamente nei due casi: il test non ha rilevato malfunzionamenti e si può passare alla entry successiva; altrimenti è presente di una discordanza di informazioni ed è necessario segnalarla: il test ha rilevato un malfunzionamento.

Capitolo 2
Analisi degli strumenti

In questo capitolo si passa in rassegna lo stato dell'arte sugli strumenti di supporto all'attività di test in applicazioni web. I due strumenti open source più volte menzionati, HtmlUnit e Selenium, vengono esaminati e messi a confronto attraverso la valutazione di un particolare caso di studio (test di unità sull'applicazione di Ateneo Web Docenti). Prima di progettare un qualunque test su una pagina web è necessario raccogliere e conservare le informazioni che la caratterizzano. Ciascuno dei due strumenti attua una tecnica completamente diversa per estrarre dal web tali informazioni: Selenium provvede a questo mediante la fase di Capture, HtmlUnit invece, lo fa per sua natura, poiché svolge la funzione di *wrapper*. Poi come vengono eseguiti i test? Come si vedrà, in Selenium basta replicare (fase di Replay) le azioni registrate, mentre in HtmlUnit bisognerà scrivere manualmente righe di codice di test e poi mandarle in esecuzione. I risultati, tuttavia, sono visibili in entrambi allo stesso modo, usando l'ambiente di test JUnit su Eclipse.

2.1 HtmlUnit

HtmlUnit è un framework che appartiene alla famiglia XUnit. Nasce per automatizzare il *test di unità* per pagine web HTML ed è scritto in linguaggio Java.

HtmlUnit

Figura 5: Logo di HtmlUnit

Della stessa famiglia fa parte anche *HttpUnit*, ma la differenza sostanziale tra i due sta nel fatto che HttpUnit lavora essenzialmente su protocolli HTTP, senza collaborare con un browser, mentre invece HtmlUnit, essendo un wrapper, lavora di pari passo col browser. Un *wrapper* è un software che estrae dalle pagine web scritte in HTML le informazioni sulla struttura, sintassi e semantica della pagina. Queste potranno poi essere rappresentate graficamente, nel formato XML, mediante l'albero DOM (Document Object Model). Nell'albero DOM ad ogni tag della pagina corrisponde un nodo. Questa operazione è possibile grazie al parsing della pagina, ovvero l'interpretazione del linguaggio X/HTML e la successiva traduzione in nodi. In questo modo, l'albero contiene i dati di partenza, da cui si estrarranno solo quelli di interesse (Figura 6). I tag più comunemente usati sono: <TITLE>, <TABLE>, <TR>, <TD>, <P>, <UL>.

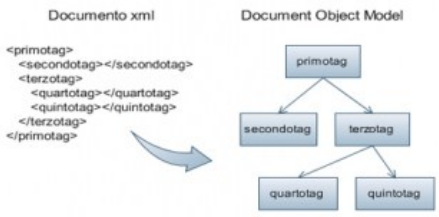


Figura 6: Realizzazione dell'albero DOM

Dettagli su HtmlUnit

- *Documentazione e supporto:* il framework ha un sito appositamente dedicato (http://htmlunit.sourceforge.net/) con relativa documentazione e qualche esempio di utilizzo. HtmlUnit è distribuito con licenza Apache 2.0;
- Ambienti Operativi supportati: Windows, Linux, Macintosh;
- *Versione*: la versione attuale è la 2.9 del 09/09/2011;
- *Processo di installazione*: il file scaricabile è un .zip (htmlunit-2.9) che si compone di due sottocartelle: una contenente la documentazione (apidocs), l'altra contenente le librerie (lib) in formato .jar (archivi Java) che si importano in progetti di test Java. L'ambiente di sviluppo utilizzato è Eclipse INDIGO.
- *Funzionalità*: HtmlUnit supporta la *fase di Replay* delle pagine web, basandosi

sulle informazioni raccolte mediante l'albero DOM;

- *Tipi di test supportati:* HtmlUnit nasce per automatizzare il *test di unità* (come dice la parola stessa "unit"), ma è usato robustamente anche per il *test black box,* perché le informazioni utili della pagina sono raccolte dal wrapper, quindi al programmatore non è necessario conoscere la struttura del programma per scrivere i test;

- *Grado di automazione*: l'utente costruisce in forma completamente autonoma i casi di test usando i package messi a disposizione da HtmlUnit. E' necessaria una conoscenza di base del linguaggio Java, dell'ambiente di sviluppo Eclipse (come creare un progetto, inserire file esterni…) e dell'uso di JUnit, un valido ambiente di test per linguaggio Java e plug-in Eclipse. JUnit è scaricabile in formato jar dal sito https://github.com/KentBeck/junit/downloads e naturalmente va incluso al progetto finale.

2.2 Selenium

Figura 7: Logo di Selenium

Selenium è un framework per l'automazione del test di applicazioni web attraverso *le fasi di Capture e Replay*. E' molto flessibile perché è possibile generare del codice (in automatico) anche senza la conoscenza di un linguaggio specifico, aspetto questo che lo differenzia notevolmente da HtmlUnit. Il linguaggio di default usato da Selenium alla creazione del test è *Selenese*, che si serve di comandi nel formato HTML. Anche Selenium è distribuito con la licenza Apache 2.0. I tool principali che questo strumento comprende sono:

- *Selenium IDE*: un ambiente di sviluppo integrato come plug-in per Firefox per la creazione rapida di singoli test o più test (suite). I test creati possono essere salvati in differenti formati ed "editati" direttamente dall'IDE per poter essere eseguiti in qualsiasi momento;
- *Selenium RC* (Remote Control): un server Java che permette di scrivere e manovrare i test direttamente da codice (sono supportati svariati linguaggi di

programmazione: C#, Java, PHP, …) e di eseguirli su diversi tipi di browser;

- *Selenium Grid*: un sistema per "espandere" Selenium RC e permettere l'esecuzione dei test in parallelo su server multipli.

Si considerano in particolare Selenium IDE e Selenium RC:

Dettagli su Selenium IDE

- *Documentazione e supporto*: il sito seleniumhq.org presenta un'ampia documentazione, con esempi e tutorial. Il supporto al progetto è molto attivo: nel sito sono presenti anche un blog ed un forum, attraverso cui gli sviluppatori e la community interagiscono;
- Ambienti operativi supportati: Windows, Linux, Macintosh;
- *Versione*: la versione attuale è la 1.8.1, dell'01/06/2012;
- *Processo di installazione*: il framework si installa come qualsiasi altra estensione di Firefox, in modo semplice ed immediato. E' subito disponibile in FireFox nel menu *Firefox->Sviluppo Web –> Selenium IDE* e da qui lo si può lanciare;
- *Funzionalità*: Selenium IDE permette la creazione dei casi di test, scritti in un linguaggio di script detto *Selenese* che si compone di comandi HTML in forma principalmente automatica, attraverso la fase di Capture e quella di Replay.
- *Tipi di test supportati*: Selenium è usato principalmente per automatizzare il *test delle interfacce utente* in applicazioni web, quindi si occupa di *test di unità*.

Una caratteristica singolare di Selenium è la possibilità di eseguire *test su macchine remote*, come già annunciato in Selenium Grid. Un server centrale, detto hub, è collegato a un insieme di server minori. Per ottenere l'accesso al browser, questi ultimi devono contattare l'hub, che ripartisce le risorse del sistema a divisione di tempo fra tutti i server. Ciò accade talmente velocemente da far percepire a ciascun server di essere l'unico utente ad utilizzare l'hub. In questo modo, si possono eseguire test in parallelo su diverse macchine.

- *Grado di Automazione*: il funzionamento è semplice ed intuitivo, a partire dalla fase di installazione, alla creazione dei casi di test fino alla loro esecuzione e lettura dei risultati.

Dettagli su Selenium RC

- *Documentazione e supporto / Ambienti operativi supportati*: valgono le considerazioni appena fatte per Selenium IDE;
- *Versione*: la versione attuale è la 2.24, dell'01/06/2012;
- *Processo di installazione*: il file scaricabile è un file di estensione .jar. che va incluso nel progetto Java che successivamente si crea. In questo, Selenium RC è molto simile ad HtmlUnit;
- *Funzionalità*: Affinché Selenium RC esegua correttamente il test c'è bisogno di *instaurare una connessione ad server*. Di default il server Selenium RC rimane in ascolto sulla *porta 4444*, ma se la connessione dovesse non stabilirsi sarà necessario specificarlo nuovamente, operando da linea di comando.

2.3 Descrizione del caso di studio

Il caso di studio che si considera è relativo all'applicazione di Ateneo Web Docenti. *Web Docenti* è l'applicazione web a cui sia studenti che docenti dell'Ateneo Federico II possono far riferimento per vari motivi: gli studenti usano l'applicazione per effettuare ricerche relative ad un particolare docente (mediante cognome) oppure ad un particolare insegnamento; i docenti la utilizzano per accedere alla propria area riservata su cui lavorare alle attività didattiche e di ricerca. L'applicazione è aggiornata alla versione 1.8, datata 15/03/2010 e si trova alla pagina web: https://www.docenti.unina.it/Welcome.do.

Si vuole creare un *semplice caso di test delle unità*: ci si immedesima in un studente che vuole ricercare (nella sezione insegnamenti) lo specifico insegnamento "Ingegneria del Software (9 CFU)". Di tutte le istanze trovate (che sono organizzate in una tabella che appare allo studente poco dopo) si considera quella relativa al particolare docente Tramontana Porfirio. Si vuole successivamente entrare nella pagina web del professore e visualizzare gli appelli d'esame disponibili. Si utilizzano, a tale scopo, gli strumenti appena descritti.

2.3.1 Caso di studio con HtmlUnit

In HtmlUnit l'utente costruisce i casi di test usando JUnit all'interno di Eclipse. E' sufficiente scrivere una sottoclasse della classe Java TestCase ed aggiungere i metodi di test. E' importante notare che tutti i metodi di test devono iniziare con la parola chiave "test". Nel dettaglio, si crea un nuovo progetto Java, al quale si aggiunge una nuova classe che è organizzata in questo modo: contiene il costruttore, i metodi di inizio e fine test, rispettivamente `setUp()` e `tearDown()`, il metodo driver proprio del test, `testIinsegnamentoIngSw()` e una funzione principale che richiama l'esecuzione del test sulla classe specificata. Dopo aver importato nel file .java le classi necessarie ed aver incluso nel progetto tutti i file .jar esterni relativi ad HtmlUnit e a JUnit, si passa alla stesura del codice. Java fa uso del *costrutto assert* che permette di testare le proprie applicazioni. Un assert è una condizione necessaria che lo sviluppatore ritiene di dover verificare. Qualora non sia verificata, il programma termina. Nel caso in esame, le verifiche sulle pagine web che si percorrono dall'inizio della ricerca dell'insegnamento "Ingegneria del Software" fino alla visualizzazione degli appelli d'esame del prof. Tramontana nel codice vengono implementate a mezzo dei metodi *assertEquals(valore, espressione)* ed *assertTrue(condizione)*. Il metodo *assertEquals* verifica che il primo parametro (valore) sia uguale al secondo (espressione): se è così, allora si passa all'istruzione seguente. Analogamente, il metodo *assertTrue* verifica che la condizione booleana specificata sia vera. Il codice MyTest.java è riportato sotto:

```
import junit.framework.TestCase;
import com.gargoylesoftware.htmlunit.*;
import com.gargoylesoftware.htmlunit.html.*;
import java.util.*;
public class MyTest extends TestCase {
private WebClient m_webclient;
private static final String SITO="https://www.docenti.unina.it/Welcome.do";
String lastURL="https://www.docenti.unina.it/Welcome.do";
public  MyTest() throws Exception{
       System.out.println("Costruttore");
       m_webclient = new WebClient();
       m_webclient.setThrowExceptionOnScriptError(false);//se trova warning
       m_webclient.setThrowExceptionOnFailingStatusCode(false);//se trova errori
}
public void setUp() throws Exception {
System.out.println("START !");
MyTest mytest=new MyTest();
}
```

```
            private void verifyHomePageHtmlPage(HtmlPage p_LastPageView){
            assertEquals("Docenti.unina.it", p_LastPageView.getTitleText());
            }
            private void verifyFirstPage(HtmlPage p_LastPageView){
            assertTrue(p_LastPageView.asText().contains("Risultato della ricerca"));
            assertTrue(p_LastPageView.asText().contains("Insegnamenti trovati"));
            assertTrue(p_LastPageView.asText().contains("Cod."));
            assertTrue(p_LastPageView.asText().contains("31680"));
            assertTrue(p_LastPageView.asText().contains("INGEGNERIA DEL SOFTWARE (9 CFU)"));
            assertTrue(p_LastPageView.asText().contains("TRAMONTANA"));
            assertTrue(p_LastPageView.asText().contains("PORFIRIO"));
            }
            private void verifyProfSite(HtmlPage p_LastPageView){
                    assertTrue(p_LastPageView.asText().contains("TRAMONTANA"));
            }
            private void findExams(HtmlPage p_LastPageView){
                assertTrue(p_LastPageView.asText().contains("Appelli esame"));
                assertTrue(p_LastPageView.asText().contains("31680"));
            assertTrue(p_LastPageView.asText().contains("INGEGNERIA DEL SOFTWARE (9 CFU)"));
            }
            private void findExamsIngSoft(HtmlPage p_LastPageView){
            assertTrue(p_LastPageView.asText().contains("Appelli d'esame"));
            assertTrue(p_LastPageView.asText().contains("31680"));
            assertTrue(p_LastPageView.asText().contains("INGEGNERIA DEL SOFTWARE (9 CFU)"));
            assertTrue(p_LastPageView.asText().contains("N46"));
            assertTrue(p_LastPageView.asText().contains("INGEGNERIA INFORMATICA"));
            assertTrue(p_LastPageView.asText().contains("Visualizza gli appelli"));
            }
            private void ViewExamsIngSoft(HtmlPage p_LastPageView){
            assertTrue(p_LastPageView.asText().contains("Appelli con prenotazioni SEGREPASS"));
            }
            private void  ViewTableExamsIngSoft(HtmlTable p_table){
                    final List rows = p_table.getRows();
                    assertTrue(!rows.isEmpty());
                     for (final Iterator rowIterator = rows.iterator(); rowIterator.hasNext(); ) {
                    final HtmlTableRow row = (HtmlTableRow) rowIterator.next();
                            System.out.println("Trovata una riga nella tabella");
                            final List cells = row.getCells();
                            assertTrue(!cells.isEmpty());
            for (final Iterator cellIterator = cells.iterator(); cellIterator.hasNext();){
             final HtmlTableCell cell = (HtmlTableCell) cellIterator.next();
             System.out.println(" RIGA "+row+" : " + cell.asText()); }   }
}
    public void testInsegnamentoPage() throws Exception{
        HtmlPage a_LastPageView= (HtmlPage) m_webclient.getPage(SITO);// Pagina iniziale
        verifyHomePageHtmlPage(a_LastPageView);
        //cerca insegnamento
        final HtmlForm form1 = a_LastPageView.getFormByName("cercainsegnamento");
        final HtmlSubmitInput submit=form1.getInputByValue("avvia ricerca");
        final HtmlTextInput insegnamento=form1.getInputByName("insegnamento");
        insegnamento.setValueAttribute("Ingegneria del Software"); //INSERITO DALL'UTENTE
        a_LastPageView=submit.click();
        verifyFirstPage(a_LastPageView);
        //cerco il sito del prof TRAMONTANA
        final HtmlForm form2=a_LastPageView.getForms().get(8);//'accesso diretto array di form
        final HtmlImageInput immagine=(HtmlImageInput)form2.getInputByName("Submit");
        final HtmlPage page3=(HtmlPage)immagine.click();
        verifyProfSite(page3);
        //cerco appelli
        final HtmlAnchor linkAppelli = page3.getAnchorByText("APPELLI");
        final HtmlPage page4=(HtmlPage)linkAppelli.click();
        findExams(page4);
        //Cerco gli appelli INGEGNERIA DEL SOFTWARE (9 CFU)
        final HtmlAnchor anchor=page4.getAnchors().get(22);//il mio anchor è il n. 22 nella pagina 4
        final HtmlPage page5=(HtmlPage)anchor.click();
        findExamsIngSoft(page5);
        //Visualizzo gli appelli INGEGNERIA DEL SOFTWARE (9 CFU)
        final HtmlAnchor LinkVisualizza=page5.getAnchors().get(19);//il mio anchor è il n.19 pagina 5
        final HtmlPage page6=(HtmlPage)LinkVisualizza.click();
        ViewExamsIngSoft(page6);
        final HtmlTable table = (HtmlTable)page6.getHtmlElementById("appello");
         ViewTableExamsIngSoft(table);
    }
    public void tearDown() {
         System.out.println("STOP !");
         m_webclient.closeAllWindows();
    }
```

```
    public static void main(String[] args) throws Exception {
        junit.textui.TestRunner.run(MyTest.class);
    }
}
```

Per eseguire il test scritto si seleziona dalla barra del menù di Eclipse: *Run->Run As-> JUnit Test.* Appare un pannello con una barra che si colora di verde se il test ha funzionato correttamente (dopo aver corretto eventuali errori). La Figura 8 mostra l'esecuzione corretta del test da Eclipse:

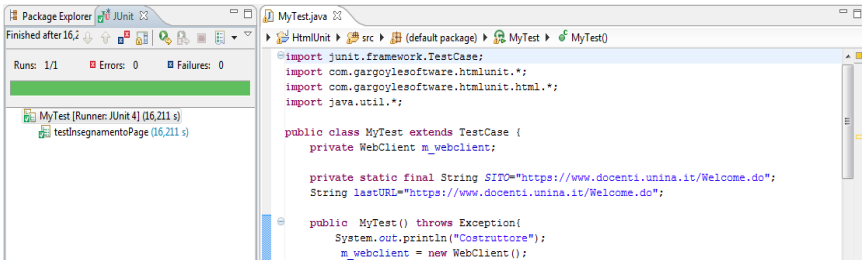
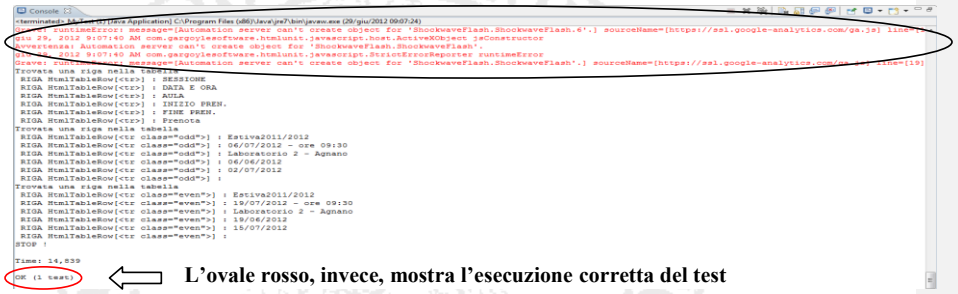


Figura 8: Esecuzione del test con HtmlUnit

HtmlUnit dunque è uno *strumento molto potente*, perché essendo anche wrapper riesce a cogliere tutti gli aspetti sintattici e semantici della pagina HTML e a rilevare così la presenza di malfunzionamenti derivanti dalla cattiva struttura della pagina. Anche nel caso in esame si è rilevata una serie di malfunzionamenti di questo tipo, riscontrabili mandando in esecuzione l'applicazione Java (*Run->Run As->Java Application*), come mostra l'ovale nero di Figura 9.



L'ovale rosso, invece, mostra l'esecuzione corretta del test

Figura 9: Esecuzione del test come applicazione

2.3.1 Caso di studio con Selenium

Per svolgere il test si utilizza la versione 13 di Firefox e 1.0.6 di Selenium IDE. Una volta installato e aperto Selenium su Firefox, l'interfaccia utente si presenta come in Figura 10. Il "pallino rosso" premuto indica che la registrazione è attiva.

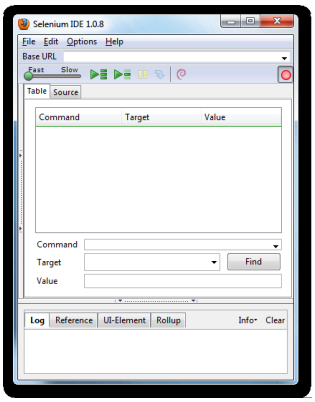


Figura 10: Selenium IDE

Quindi ogni interazione con il browser sarà registrata e potrà essere riprodotta in modo automatico rieseguendo il test. Nella barra URL si inserisce l'URL della pagina da cui ha inizio il test: https://www.docenti.unina.it/. Anche Selenium utilizza le asserzioni. Qui si distinguono in tre categorie: *assert, verify e wait.* Nel caso in esame si utilizzano solo le prime due *(assert e verify)*. Queste si comportano in modo simile ma hanno una sostanziale differenza: se un assert fallisce, il test termina immediatamente, riscontrando un fallimento; se invece è un verify a fallire, il test continua l'esecuzione e rileva un failure solo alla fine. La Figura 11 mostra la situazione in Selenium IDE, dopo aver completato la cattura delle interazioni (fase di Capture). I comandi utilizzati per la cattura sono stati: *open, type, clickAndWait, verifyTable, verifyTextPresent, assertTextPresent*. Tralasciando i comandi *open, type, clickAndWait* generati automaticamente da Selenium in fase di Capture e il cui significato è autoesplicativo, ci si concentra maggiormente sugli altri tre, per i quali è stato necessario l'intervento del tester: in questo modo il tester progetta i casi di test con l'aiuto di Selenium IDE, quindi in modo parzialmente automatizzato.

Command	Target	Value
open	/Welcome.do	
type	name=insegnamento	Ingegneria del software
clickAndWait	xpath=(//div[@id='inputDiv']/form/table/tbody/tr[2]/td[2]/a/span)[2]	
assertTextPresent	Risultato della ricerca	
assertTextPresent	Insegnamenti trovati	
assertTextPresent	Cod.	
verifyTable	id=insegnamento.9.0	31680
verifyTable	id=insegnamento.9.1	INGEGNERIA DEL SOFTWARE (9 CFU)
verifyTable	id=insegnamento.9.2	TRAMONTANA
verifyTable	id=insegnamento.9.3	PORFIRIO
clickAndWait	xpath=(//input[@name='Submit'])[9]	
assertTextPresent	TRAMONTANA	
clickAndWait	link=APPELLI	
assertTextPresent	Appelli esame	
verifyTable	id=insegnamento.1.0	31680
verifyTable	id=insegnamento.1.1	INGEGNERIA DEL SOFTWARE (9 CFU)
clickAndWait	css=img[alt="procedi"]	
assertTextPresent	Appelli d'esame	
assertTextPresent	31680	
assertTextPresent	INGEGNERIA DEL SOFTWARE (9 CFU)	
verifyTable	css=table.bordata.1.2	Visualizza gli appelli
clickAndWait	link=Visualizza gli appelli	
verifyTextPresent	Appelli con prenotazioni SEGREPASS	

Figura 11: Comandi generati in fase di Capture

Il comando *assertTable* verifica il testo di una particolare cella della tabella; il comando *verifyTextPresent* verifica che il testo specificato appaia da qualche parte sulla pagina visualizzata dall'utente e similmente *assertTextPresent*. Comunque, la correttezza del test effettuato va dimostrata usando Eclipse. Si crea un nuovo

progetto Java in cui si importano i file .jar relativi a Selenium IDE e a Selenium RC. Si crea una nuova classe all'interno del progetto in cui si inserisce il codice generato da Selenium IDE nel formato JUnit 4. Si salva il progetto, che finalmente viene eseguito: *Run->Run As-> JUnit Test*. Dopo aver corretto eventuali errori, ha il via la riproduzione delle interazioni (fase di Replay) nella pagina testata mediante la *Command History*, una finestra situata in alto a destra nella stessa pagina che esegue sequenzialmente i comandi generati. Al termine del test, in Eclipse compare la barra verde che è simbolo del corretto funzionamento del test. I momenti "salienti" dell'esecuzione sono mostrati in Figura 12.

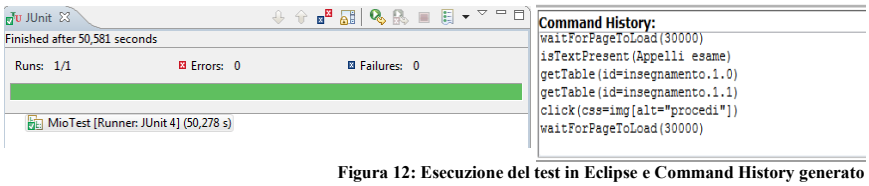


Figura 12: Esecuzione del test in Eclipse e Command History generato

Nel caso in cui il server Selenium RC non risulti avviato, Eclipse riscontra il problema e richiede l'avvio dello stesso in modo manuale operando da linea di comando. In un sistema Windows, si fa click su *Start->Esegui*. Si digita cmd e compare la finestra del Prompt dei comandi. Si imposta la linea di comando a **c:\>**, si inserisce il path in cui si trova il file selenium-servestandalone-2.24.1.jar (che è stato rinominato ss.jar) e si preme invio. Per poter stabilire la connessione, si digita: *java –jar ss.jar –port 4444*, come evidenziato in Figura 13 e si esegue il test nuovamente da Eclipse. Per velocizzare la procedura si può realizzare anche un batch file che esegue il comando in modo automatico.

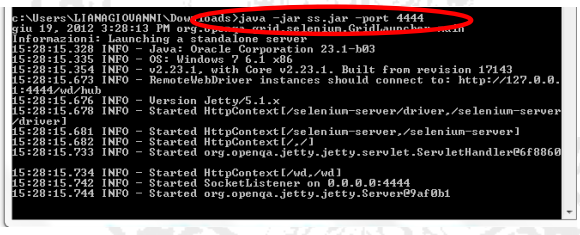


Figura 13: Risoluzione connessione da linea di comando

A seguire è riportato il codice della classe *MyTest.java*:

```
import com.thoughtworks.selenium.*;
public class MyTest extends SeleneseTestCase {
public void setUp() throws Exception {
      selenium = new DefaultSelenium("localhost", 4444, "*chrome",
      "https://www.docenti.unina.it/");
```

```
                selenium.start();
        }
        public void testInsegnamentoIngSw () throws Exception {
        selenium.open("/Welcome.do");
        selenium.type("name=insegnamento", "Ingegneria del software");
        selenium.click("xpath=(//div[@id='inputDiv']/form/table/tbody/tr[2]/td[2]/a/span[2]")
;       selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("Risultato della ricerca"));
        assertTrue(selenium.isTextPresent("Insegnamenti trovati"));
        assertTrue(selenium.isTextPresent("Cod."));
        verifyEquals("31680", selenium.getTable("id=insegnamento.9.0"));
        verifyEquals("INGEGNERIA DEL SOFTWARE (9 CFU)",
        selenium.getTable("id=insegnamento.9.1"));
        verifyEquals("TRAMONTANA", selenium.getTable("id=insegnamento.9.2"));
        verifyEquals("PORFIRIO", selenium.getTable("id=insegnamento.9.3"));
        selenium.click("xpath=(//input[@name='Submit'])[9]");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("TRAMONTANA"));
        selenium.click("link=APPELLI");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("Appelli esame"));
        verifyEquals("31680", selenium.getTable("id=insegnamento.1.0"));
        verifyEquals("INGEGNERIA DEL SOFTWARE (9 CFU)",
        selenium.getTable("id=insegnamento.1.1"));
        selenium.click("css=img[alt=\"procedi\"]");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("Appelli d'esame"));
        assertTrue(selenium.isTextPresent("31680"));
        assertTrue(selenium.isTextPresent("INGEGNERIA DEL SOFTWARE (9 CFU)"));
        verifyEquals("Visualizza gli appelli", selenium.getTable("css=table.bordata.1.2"));
        verifyTrue(selenium.isTextPresent("Visualizza gli appelli"));
        selenium.click("link=Visualizza gli appelli");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Appelli con prenotazioni SEGREPASS"));
    }
        public void tearDown()throws Exception {
        selenium.stop();}
}
```

Un rapido sguardo al codice generato da Selenium mostra la sostanziale differenza rispetto ad HtmlUnit: non è presente né il costruttore della classe MioTest né il main. Ciò significa che Selenium, oltre a generare automaticamente i metodi `setUp()`, `tearDown()` e `testIinsegnamentoIngSw()`, incorpora anche le funzionalità che invece in HtmlUnit è necessario esplicitare.

2.4 Confronto con altri strumenti

In questo paragrafo si descrivono a scopo illustrativo le caratteristiche principali di altri strumenti per il testing funzionale di applicazioni web similari a quelli già visti, classificati nella categoria degli strumenti per il *test funzionale*.

Si evidenziano analogie e differenze rispetto agli strumenti già usati.

2.4.1 FuncUnit

E' un framework open source per il *testing*

Figura 14: Logo di FuncUnit

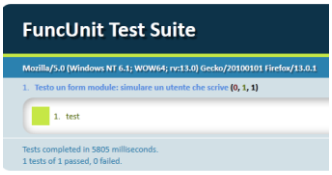


Figura 15: Visualizzazione dei risultati con FuncUnit framework

funzionale di applicazioni web. Utilizza un insieme di API basate su jQuery (nuovo tipo di libreria JavaScript) e *lavora su pagine HTML*. FuncUnit si appoggia a vere pagine web, aprendole nel browser ed interagendo con esse. I test sono scritti manualmente con *sintassi JQuery* e i comandi sono inviati dallo script funcunit della pagina attraverso una finestra popup (Figura 15). Il framework è scaricabile dal sito ufficiale e si compone perlopiù di librerie da includere al progetto Java. In alternativa, FuncUnit supporta l'IDE FuncIT (versione alpha) e funziona in modo del tutto simile a Selenium. Non è scaricabile ma viene lanciato come prova on-line: invece di inviare comandi a una finestra popup, si inviano a Selenium, il cui server restituisce il *codice JQuery* generato per FuncUnit (Figura 16). In entrambi i casi si colgono anche *somiglianze con HtmlUnit* poiché lo strumento *lavora su pagine HTML*.

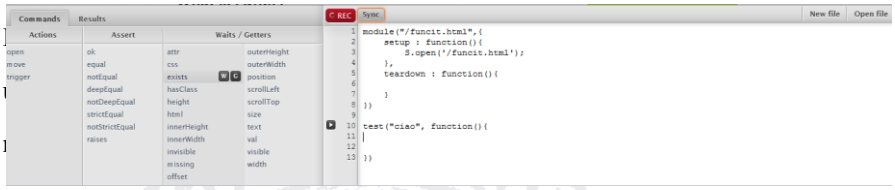


Figura 16: Visualizzazione dei risultati con FuncUnitIT

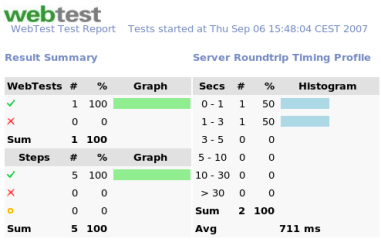
FuncUnit supporta i principali browser: Internet Explorer, Firefox, Safari, Opera, Chrome, su PC, Mac, e Linux. Il sito ufficiale dello strumento, http://www.funcunit.com/, si compone di sezioni che rimandano a link di documentazione, tutorial, forum e community.

2.4.2 Canoo WebTest

E' uno strumento open source per il *testing funzionale* di applicazioni web. E' scritto in Java e contiene una serie di *librerie Ant* per la simulazione. E' molto simile ad *HtmlUnit*, essendo anch'esso un *browser "faceless"*. I test sono scritti in una sintassi

Figura 17: Logo di Canoo Web Test

semplice, in genere XML o Groovy (linguaggio di scripting per Java), comprensibile anche da chi non è pratico di WebTest. Forse la caratteristica più significativa di Canoo WebTest è che, dopo che il test è stato eseguito da linea di comando, nel browser è visibile una *tabella* contenente tutte le informazioni che permettono di comprendere rapidamente quale sia stato il comportamento dell'applicazione durante il test e quali sono le cause di eventuali fallimenti (Figura 18). E' anche facile da estendere a seconda delle esigenze: aggiungendo le apposite librerie possibile scrivere test in Java ma il supporto Javascript non avrà la stessa qualità di un browser "normale", dato che il comportamento del browser, quando il test viene eseguito, viene soltanto simulato. Inoltre, Canoo WebTest per il momento accetta abbastanza bene il formato HTML, ma gli sviluppatori hanno fiducia in un ulteriore miglioramento futuro. Nella sua versione 3.0 si include supporto per i *test di unità*, oltre a una sempre migliore *esecuzione parallela dei test*. Ogni versione scaricabile di Canoo WebTest include lo stesso set di file: l'applicazione web che Canoo usa per testare se stesso, in formato .war, la versione, la documentazione, i file Java src e il supporto per l'integrazione Maven (supporto Java che fornisce un modello utile per la gestione e progettazione di un progetto). Il sito ufficiale dello strumento, http://webtest.canoo.com/, è ricco di informazioni utili: manuale arricchito da esempi di codice, sezione download e installazione, ultime notizie, mailing list per gli utenti e tanto altro.


webtest
WebTest Test Report Tests started at Thu Sep 06 15:48:04 CEST 2007

Result Summary

WebTests	#	%	Graph
✓	1	100	
✗	0	0	
Sum	1	100	

Steps	#	%	Graph
✓	5	100	
✗	0	0	
▫	0	0	
Sum	5	100	

Server Roundtrip Timing Profile

Secs	#	%	Histogram
0 - 1	1	50	
1 - 3	1	50	
3 - 5	0	0	
5 - 10	0	0	
10 - 30	0	0	
> 30	0	0	
Sum	2	100	
Avg		711 ms	

Figura 18: Tabella dei risultati di Canoo WebTest

Conclusioni e Sviluppi futuri

L'obiettivo di questo elaborato è stato la valutazione di alcuni strumenti open source per il testing di applicazioni web. I due strumenti principalmente utilizzati sono stati HtmlUnit e Selenium.

La prima parte della trattazione si è aperta riprendendo i concetti, propri dell'Ingegneria del Software, del testing e della sua automazione unendoli però ad un'analisi *allo stato dell'arte* degli strumenti suddetti, che usano la tecnica migliorativa del Capture and Replay.

Nella seconda parte, invece, si è focalizzata tutta l'attenzione sulla *valutazione vera e propria degli strumenti* citati e sull'*analisi del caso di studio* che li ha visti coinvolti. Dai risultati traspare che Selenium è sicuramente molto più semplice ed intuitivo da utilizzare, tanto che non è necessario che l'utente conosca alcun linguaggio di programmazione, poiché lo strumento lavora per tutta la durata della fase di testing in modo completamente automatico, generando alla fine il codice corrispondente al caso di test che verrà eseguito aprendo nel browser la Command History Piuttosto se si desidera visualizzare in forma chiara l'esito del test è necessario creare un progetto Java a cui aggiungere l'ambiente di test JUnit. Una volta mandato in esecuzione come JUnit test, l'esito sarà dato dal colore della barra (verde se il test è andato bene, rosso viceversa). L'approccio con HtmlUnit, invece, è sicuramente più impegnativo perché si suppone una conoscenza, seppure di base, del linguaggio Java e degli strumenti di cui bisogna necessariamente fare uso per l'esecuzione del test (gli ambienti Eclipse e JUnit). Come si è visto, i test devono essere scritti inevitabilmente dall'utente in un progetto Java e solo successivamente

potranno essere eseguiti in modo automatico.

Per finire, si è effettuato un ulteriore confronto con altri due strumenti di web testing: FuncUnit e Canoo WebTest, mettendone in evidenza le somiglianze rispettivamente con Selenium e con HtmlUnit, ma comunque valutandoli nelle loro peculiarità.

Allo stato dell'arte tutti gli strumenti descritti risultano molto utili ed efficienti.

Risulta spontaneo, a questo punto, pensare all'equivalente *commerciale* di tali strumenti. Possono esistere notevoli vantaggi nell'utilizzo di uno strumento open source rispetto ad uno commerciale, ma tutto dipende dal contesto in cui li si usa.

Sicuramente, uno dei motivi che spinge a scegliere uno strumento open source è il fatto che non costa nulla, che lo si "trova" senza troppa fatica sul web e che spesso, se non si hanno troppe pretese, consente di ottenere risultati soddisfacenti.

Al contrario, aziende, organizzazioni ed altri enti di un certo livello potrebbero preferire uno strumento commerciale, perché è in grado di fornire un maggiore supporto e una maggiore qualità e manutenzione rispetto all'analogo open source.

Questo però non significa che il software open source passi in secondo piano, anzi. *Oggigiorno la sempre crescente attenzione ai costi amplifica l'interesse del mercato verso il software open source*. Ma è necessario comunque che si scelgano soluzioni consolidate e ampiamente supportate. Grazie allo spirito di condivisione, di sviluppo e di ridistribuzione dei team di sviluppatori sparsi in tutto il mondo, tutto questo è possibile.

Il software open source è quindi un'arma di conoscenza potentissima per tutti.

Bibliografia e siti web

[1] I. Sommerville, 2007, "Ingegneria del Software, Ottava Edizione"

[2] R. Pressman, 2008, "Principi di Ingegneria del Software, 5a Edizione"

[3] C.Ghezzi, M.Jazayeri, D.Mandrioli, 2004, "Ingegneria del Software - Fondamenti e Principi, 2a edizione"

[4] http://seleniumhq.org/

[5] http://htmlunit.sourceforge.net/

[6] https://www.docenti.unina.it/Welcome.do

[7] http://www.softwareqatest.com/qatweb1.html

[8] http://www.funcunit.com/

[9] http://webtest.canoo.com/webtest/manual/WebTestHome.html

Ringraziamenti

In questo mio percorso universitario ho imparato tante cose e la maggior parte di esse forse non le ho apprese dai libri.

La cosa più bella che l'università mi ha insegnato è stata conoscere la soddisfazione che si prova quando si supera un ostacolo, piccolo o grande che sia: la sensazione di forza che ne deriva è immensa ed è proprio quella forza che permette di cimentarsi al meglio nella prossima sfida. Così, senza quasi rendermene conto, è arrivato il momento di impegnarmi nell'ultimo giro di giostra, nell'ultima sfida, quella che alla fine dona la gioia più grande perché conduce al traguardo tanto ambito!

Quanta vita che è racchiusa in questo momento, quanta passione investita nel fare le cose, quante cadute e quante riprese vissute...

Grazie a tutte le persone che mi hanno accompagnata in questo percorso: a chi ha scoperto il mondo universitario condividendo questi anni con me, a chi ha riso con me, a chi mi ha dato sostegno quando ne avevo più bisogno, a chi mi è stato da guida quando avevo nella testa dubbi e paure, a chi mi è stato sempre accanto e mi ha donato la forza e la determinazione di persistere fino ad oggi.

Un ringraziamento speciale va al mio relatore, il professore Porfirio Tramontana, sempre disponibile a darmi preziosi consigli, e al professore Antonio Iorio, guida conclusiva nel mio lavoro di tesi.

Grazie di cuore!

Liana