Università degli Studi di Firenze
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Triennale in Informatica

Web Testing: Teoria ed Applicazioni

Laureando:
Claudio Tortorelli

Relatore:
Prof. Pierluigi Crescenzi

Correlatore:
Carlo Salinari

29 Settembre 2003

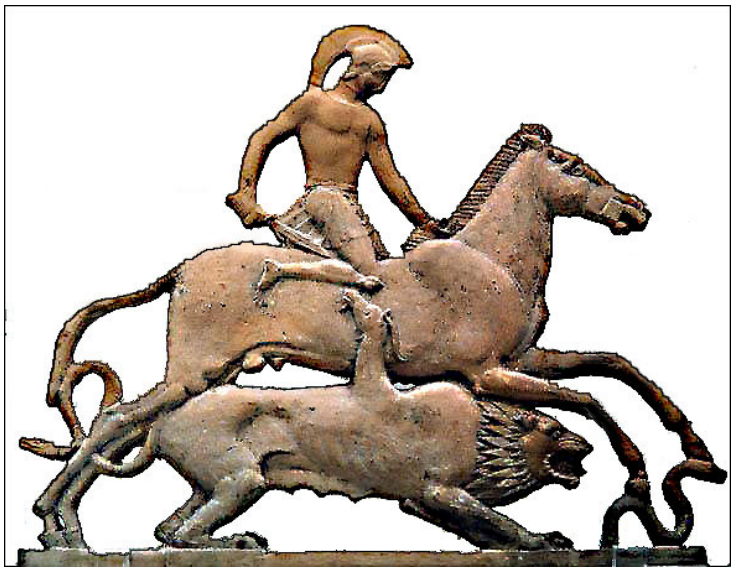

Figura 1: Bellerofonte lotta con la Chimera

Indice

Introduzione **6**

I Software e web testing 9

1 Il software testing **10**
1.1 Perché c'è bisogno di software testing 10
1.2 Definizioni e concetti di base 12
1.3 Gli obiettivi del software testing 15
1.4 Rilevare un bug 16
1.4.1 Alcune tipiche complicazioni 17
1.5 Le cinque fasi operative del software testing 18
1.5.1 Modellazione dell'ambiente di lavoro 18
1.5.2 Selezione degli scenari di test 19
1.5.3 Esecuzione e valutazione degli scenari 20
1.5.4 Correzione degli errori rilevati 22
1.5.5 Misurazione dei progressi nel testing 23
1.6 eXtreme Programming: il testing come filosofia 24
1.7 ROC: se i bug sono considerati inevitabili 25
1.8 Conclusioni 27

2 Applicazioni di rete e Web Service **28**
2.1 Il paradigma Client/Server 28
2.2 Il Web 31
2.3 I servizi di rete 32
2.4 I web service 33
2.4.1 Architettura dei web service 34
2.5 Conclusioni 35

3 Il web testing **36**
3.1 Cosa si deve testare 36

3.2 Una panoramica generale 38
3.2.1 Acceptance testing 38
3.2.2 Feature-level testing 38
3.3 Analisi di alcune categorie di web testing 40
3.3.1 User Interface test 40
3.3.2 Functional test 42
3.3.3 Database test 43
3.3.4 Installation test 44
3.3.5 Configuration e compatibility test 46
3.3.6 Performance, load e stress test 47

4 Breve rassegna del software di web testing 49

II Web testing con Bellerofonte 52

5 Presentazione del software 53
5.1 Descrizione ad alto livello 53
5.1.1 Dal mito al software 53
5.1.2 Cosa è Bellerofonte 54
5.1.3 Gerarchia dei test eseguibili 55
5.2 Dettagli implementativi 56
5.2.1 Linguaggio e strumenti di sviluppo 56
5.2.2 Utilizzo di pacchetti esterni 57
5.2.3 Identificazione delle classi 58
5.2.4 Analisi dei metodi più significativi 62
5.2.5 Rassegna dei test base implementati 63
5.3 Estendere Bellerofonte 65
5.3.1 Come aggiungere funzionalità ex-novo 65
5.3.2 Possibili sviluppi futuri 70

6 Manuale d'uso 71
6.1 File di programma 71
6.2 Eseguire Bellerofonte 74
6.2.1 Preparare il file opzioni 74
6.2.2 Preparare il file test 75
6.2.3 Gestire i risultati 77
6.2.4 La console interattiva 79
6.2.5 Esempi pratici 79

III La piattaforma Orbilio: sviluppo e testing 84

7 Breve descrizione di Orbilio 85

8 Testing di Orbilio con Bellerofonte 88
8.1 Un semplice esempio di Installation test 89
8.2 Conclusioni 92

Bibliografia 94

Indice Analitico 97

Introduzione

Una delle parti sicuramente più insidiose e dispendiose del processo di sviluppo del software è sicuramente quella del **testing**. Si tratta però di un passaggio inevitabile nella produzione di software di qualità. Che si usi un modello di sviluppo *a cascata* piuttosto che uno *a spirale* il testing offre comunque occasione di verifica della bontà dei risultati raggiunti, in relazione ai requisiti richiesti.

Con il termine "software testing" si fa riferimento ad un'ampia gamma di operazioni volte a dimostrare la validità funzionale e logica del software prodotto. Anche quando non è possibile dimostrare in modo generale l'idoneità di un certo asserto, è utile trovare dei test empirici che mettano in evidenza eventuali carenze del software, in un dominio più ristretto ma specifico di casi.

Ma cosa si intende per "utilità'' del testing? Cosa dovrebbe spingere non solo la grande "software house" ma anche il "laboratorio artigianale" ad addentrarsi nella faticosa e spesso lunga attività di ricerca di errori? L'attività di testing è da intraprendere per ogni categoria di software?

Nel corso di questa tesi si tenterà di rispondere a queste domande sia con constatazioni teoriche sia con esempi pratici.

É noto che, per quanto sia esperto ogni singolo sviluppatore di un gruppo di lavoro, il "*bug* software'' si verifica puntualmente, per fattori intrinseci dell'attività di programmazione. Ad esempio, i tre stadi di analisi, progetto e sviluppo non si svolgono quasi mai nel modo lineare e consecutivo che si vorrebbe: l'attività dei singoli sviluppatori tende a sovrapporsi, generando incomprensioni e fasi di sviluppo "caotico'', con conseguente alta probabilità di errori logici e semantici.

Un altro aspetto che può influenzare negativamente la qualità di un prodotto è la scarsa conoscenza del contesto nel quale il software opererà, delle attività e delle procedure che dovrà supportare ed integrare. Esiste software destinato ad utenti generici e software destinato a particolari classi di utenti, con esigenze che possono essere del tutto estranee alla cultura del progettista o dello sviluppatore. Vi sono varie tecniche per colmare queste lacune e for-

nire un prodotto ottimizzato per una particolare funzione. Ciononostante è facile trascurare qualche particolare ed andare incontro a malfunzionamenti.

L'attività di controllo e refactoring di un software è spesso scarsamente presa in considerazione ed è condotta con metodi non idonei e poco efficienti. Eppure in molti casi il successo di un software può essere determinato proprio dalla cura dei "dettagli". Aspetti apparentemente secondari, quali la manutenibilità, la robustezza, l'affidabilità e l'usabilità, hanno oneri alti sia in termini monetari che di tempo (e si sa che spesso un progetto ha carenza di entrambe queste risorse). Per una buona attività di verifica sono necessari appositi gruppi di lavoro (interni ed esterni), lo sviluppo o l'acquisto di software specifico e la continua ricerca di nuove tecnologie. Perciò è più facile e meno dispendioso puntare ai soli requisiti primari di un software, ovvero l'efficienza e la correttezza. Questo approccio "al risparmio" porta vantaggi economici e temporali nel breve periodo, mentre è dimostrato l'inverso per il lungo periodo. Infatti, in progetti di lunga durata, un'attività sistematica ed ordinata di testing porta ad ottenere software molto più valido, usabile, facilmente riutilizzabile e manutenibile. Qualità che fanno la differenza in un mercato ormai sempre più saturo e competitivo.

Per ottenere la correttezza formale e semantica del proprio software vi sono molte attività da intraprendere, ognuna rivolta ad un particolare aspetto del programma. Non sempre è necessario (anzi, è praticamente impossibile) che un software rispetti tutti i canoni delle "buone qualità'', ma almeno sulle funzionalità chiave occorre fornire adeguate garanzie. Per farlo è possibile affidarsi a *versioni beta*, simulazioni, *usability tests*, euristiche, analisi white e black box oppure ad un'oculata combinazione di queste metodologie.

Ciò che vale per il software "tradizionale'' è valido anche per quello di rete, che però si porta dietro tutta una serie di problematiche aggiuntive ampiamente descritte dalla letteratura del settore. Chi progetta servizi di rete e siti web più o meno dinamici, è tenuto ad offrire un servizio sicuro, corretto ed ottimizzato. In poche righe si può affermare che un'applicazione web:

- è spesso rivolta ad utenti dalle competenze e dagli interessi più disparati (*Internet*) ma può anche essere usata da un gruppo circoscritto ed omogeneo di utenti (*intranet*);
- può essere concepita sia per usi specifici che generici;
- può includere aspetti critici e di sicurezza;

- deve supportare tecnologie in continua evoluzione;
- può comprendere contenuti multimediali;
- è spesso in continuo aggiornamento;
- è pensata per un funzionamento distribuito, che coinvolge macchine eterogenee e topograficamente distinte moltiplicando le cause di possibili malfunzionamenti.

Da questa sintetica rassegna di peculiarità emerge l'importanza di una buona attività di testing per le applicazioni web. Attività che in questo contesto è chiamata **web testing**.

Nei capitoli che seguono sarà fatta un'analisi più dettagliata del mondo del software testing tradizionale. Si passerà poi ad esaminare brevemente in cosa consiste il software di rete con un accenno ai più specifici web service. Saranno quindi osservate più da vicino le categorie di web testing e, in conclusione della prima parte, sarà presentato qualche esempio di software per web testing attualmente sul mercato.

Nella seconda parte si descriverà Bellerofonte, il software di web testing realizzato nell'ambito di questa tesi, mentre nella terza ed ultima si applicherà il suddetto software ad un caso di studio concreto: la piattaforma per e-learning **Orbilio**.

Parte I

Software e web testing

Capitolo 1

Il software testing

L'argomento sul quale si focalizzerà l'attenzione nello sviluppo della tesi è il web testing, ma è impossibile iniziare a parlarne senza passare attraverso un'introduzione al *software testing*. Alcuni addirittura non vedono neppure una grossa distinzione tra i due tipi di testing ed in effetti il web testing può sicuramente considerarsi una specializzazione del software testing (che merita di essere considerata a parte). Questo capitolo fornirà una descrizione piuttosto generica di ciò che rappresenta oggi il software testing: non solo verifiche empiriche, ma anche e soprattutto strategie, tecniche di programmazione, esperienza, innovazioni tecnologiche, investimenti economici, profonde competenze informatiche ed in generale ogni espediente in grado di controllare la qualità e l'affidabilità del software. Tutto ciò è software testing e di conseguenza anche web testing, il quale comprende inoltre problematiche e necessità proprie delle applicazioni di rete. É ritenuto quindi un buon approccio all'argomento quello che prende in considerazione il web testing come "caso particolare" di software testing, approfondendo gli aspetti in cui maggiormente si caratterizza.

1.1 Perché c'è bisogno di software testing

Nell'introduzione a questa tesi si sono fatti accenni alle necessità dell'informatica moderna ed alle cause dei cosiddetti bug. Va a questo punto spiegato perché le aziende hanno l'obbligo di combattere i bug. In sintesi si possono elencare le seguenti motivazioni:

- i sistemi e le applicazioni si trovano a gestire dati ed operazioni sem-

pre più critiche, visto il livello di informatizzazione raggiunto in ogni settore;

- ogni errore ed ogni ritardo ha un costo sempre più elevato per le aziende come per gli utenti finali;
- l'immagine di un'azienda è fortemente influenzata dalla qualità dei suoi prodotti (vale per qualsiasi settore produttivo);
- come per altri tipi di tecnologie anche per l'informatica si sta verificando un processo di "standardizzazione della qualità''. Gli utenti, divenuti ormai numerosi, trascorsi i primi anni "dell'alfabetizzazione", si attendono un certo livello qualitativo, al di sotto del quale non sono disposti a scendere.

Si è citato, tra le altre buone ragioni per applicare il testing, l'elevato costo di ogni bug. Ma quanto incide realmente in termini economici il fenomeno dei bug sul mondo dell'informatica? Secondo stime pubblicate sull'*IBM System Journal* [7], dal 50% al 70% delle risorse per lo sviluppo delle applicazioni industriali è destinato oggigiorno a coprire i costi delle varie attività di verifica, testing e debugging.

Se agli albori dell'informatica correggere un bug era relativamente facile perché bastava intervenire su pochi *mainframe* nei quali si accentravano le attività di tanti terminali, dagli anni '80, con lo sviluppo di pc e portatili isolati, applicare *patch* ed aggiornamenti di correzione è diventato molto più costoso. Il successivo avvento di Internet ha poi aiutato in questo senso, riavvicinando in qualche modo la situazione a quella centralizzata dei mainframe (tramite i *download* via Web). L'aggravio per le aziende produttrici non si è però alleggerito: testare applicazioni complesse come quelle odierne (con interfaccia grafica, basate su eventi, che devono essere compatibili con vari sistemi operativi e che hanno a che fare con hardware disparato ed in continuo aggiornamento) ha un costo molto più alto rispetto al test delle applicazioni di una volta. Gli oneri sostenuti si rivelano essere però quasi sempre ripagati dalla capacità del prodotto di imporsi presso il grande bacino di utenza che oggi è costituito (teoricamente) dall'intera umanità. Il software testing, quale strumento principale nella lotta ai bug, finisce così per assumere un fondamentale ruolo strategico. L'espansione della "filosofia del testing" a tutti i livelli non è però mai stata facile: ha infatti subíto un andamento molto graduale, ha trovato (e sta trovando) opposizione (economica ed ideologica) tra sviluppatori e manager ed è tutt'ora lontana da livelli ottimali.

Benché la letteratura sul software testing sia apparsa già ai tempi dei primi computer, solo quando l'informatica è uscita dallo stretto alveo scientifico-militare per invadere massicciamente anche contesti commerciali, amministrativi ed universitari si è cominciato ad adottare sistematicamente tecniche di revisione del codice. Alla rivalutazione del testing hanno contribuito fortemente nei decenni passati la nascita di Internet e lo sviluppo di grandi progetti software, ai quali partecipano centinaia di persone con competenze diverse e per intervalli di tempo indefiniti. Internet è stata il veicolo di una vera informatizzazione di massa, ma anche la lente di ingrandimento su tutti i problemi di computazione distribuita, compatibilità hardware/software ed usabilità. D'altra parte l'abbandono di un approccio "artigianale" alla programmazione ha messo in evidenza tutte le difficoltà di coordinazione, correzione e manutenzione dei software moderni in un ambiente di interscambio in continua evoluzione. Il testing è passato così da attività marginale a mezzo indispensabile per garantire la qualità del software e quindi competitività sul mercato.

1.2 Definizioni e concetti di base

Ci sono poche ma importanti definizioni che serve conoscere prima di procedere. Anzitutto si può ricordare la *tesi di Dijkstra* che è la regola di riferimento di ogni verificatore:

Tesi 1 (di Dijkstra) *Scopo del testing è rilevare gli errori, non provare l'assenza degli stessi.*

Questa tesi può sembrare pessimista, ma è una conseguenza dell'indecidibilità del *problema della fermata*, ovvero del fatto che non è possibile decidere a priori se un software terminerà o meno la sua computazione. Infatti se la funzione

$$halt(x) = \begin{cases} 1 & \text{se } P_x(x) \text{ termina} \\ 0 & \text{altrimenti} \end{cases}$$

che stabilisce se un programma P_x si arresta (correttamente o meno) con input x fosse computabile, si potrebbe definire anche un'altra funzione totale

$$correct(x) = \begin{cases} 1 & \text{se } halt(x) = 1 \ \& \text{ non ci sono stati errori} \\ 0 & \text{altrimenti} \end{cases}$$

per la correttezza di $P_x(x)$. Purtroppo dalla Teoria della Computabilità si sa che *halt* non è computabile. Quindi, anche ammettendo che P_x sia

esente da errori, come lo si può dimostrare se il suo tempo di esecuzione non è limitato? La verifica deve avvenire necessariamente in un tempo finito, ma questo tempo può non essere sufficiente al rilevamento di errori. Non è dunque possibile definire un algoritmo generale che consenta di escludere la presenza di errori in un programma.

L'obiettivo si sposta allora sul trovare test efficaci, ovvero che rilevino la maggioranza degli errori (se ci sono), mentre la *correttezza* di un software è limitata ai soli test effettuati.

Def. 1 (Successo) *Si dice che un test* ha successo *quando rileva un errore.*

Def. 2 (Correttezza) *Dato un insieme A di test t_i per $i = 1...n$ si può affermare che un software S è corretto rispetto ad A se nessun $t_i(S)$ ha successo.*

Dal punto di vista logico l'attività di software testing si può scomporre in tre sotto attività:

1. *verifica*: viene effettuata dai programmatori o dai verificatori prima del rilascio del software (alla verifica corrisponde solitamente la fase di α-testing);
2. *validazione*: la esegue il cliente utilizzando il software, sulla base dei requisiti concordati in fase di analisi del progetto (durante la validazione si colloca in genere il β-testing);
3. *debugging*: individuazione e correzione passo per passo degli errori trovati nelle due fasi precedenti.

La verifica viene a sua volta fatta mediante:

- *analisi dinamica*, che avviene osservando l'esecuzione del software, tipicamente con strumenti automatici;
- *analisi statica*, che interessa l'analisi del codice, fatta sia manualmente che con strumenti automatici;
- *analisi mutazionale*, che viene effettuata inserendo di proposito errori nel codice con l'obiettivo di controllare la validità dei test stessi.

In questa tesi ci si interesserà particolarmente all'attività di verifica, concentrandosi sull'analisi dinamica.
Gli errori, poi, possono essere classificati come segue:

- *failure*: è un allontanamento dal comportamento previsto ovvero un malfunzionamento;
- *fault*: è una precisazione rispetto al failure, in quanto individua l'anomalia interna al programma che si riflette sul malfunzionamento;
- *bug*: è l'errore vero e proprio ovvero ciò che provoca l'anomalia e può essere logico, funzionale, fisico e così via.

Per esempio, sia dato un programma il cui requisito base è la corretta divisione tra due numeri. In seguito alla verifica con vari input si trova che in alcuni casi il programma si blocca e non restituisce la divisione (si verifica quindi un failure). Si individua allora un'anomalia nell'implementazione dell'operazione di divisione (ovvero un fault) ed in seguito ci si accorge che non viene impedita la divisione per zero (trovando dunque il bug).

Def. 3 (Fixed) *Si definisce* fissato *o* fixed *un bug rilevato durante il testing ed in seguito corretto.*

Dal punto di vista del verificatore si può parlare di

- *testing black-box* o funzionale, in cui partendo dalle specifiche e senza considerare i dettagli implementativi si valutano le reazioni del software in base a varie configurazioni di input (il software è visto come un "juke box'' per il quale l'inserimento di un gettone e la pressione di un certo tasto dovrebbero determinare l'ascolto di un dato disco);
- *testing white-box* o strutturale, che considera l'implementazione interna del programma e studia i casi di test che la mettano alla prova. Non è sempre possibile effettuare un simile testing se non si è sviluppatori;
- *testing gray-box*, cioè una mescolanza dei due punti di vista precedenti. Sia il black-box che il white-box testing se applicati singolarmente presentano delle lacune: il primo riesce faticosamente a rilevare bug nel flusso dei dati o nel controllo dei domini; il secondo ha invece difficoltà a verificare la compatibilità, l'usabilità e l'interoperabilità con altri software. Il gray-box testing "consiste in un insieme di metodi e tool derivati dalla conoscenza interna e diretta del software e dell'ambiente di utilizzo, applicati secondo un criterio black-box'' [Nguyen] [11].

Il testing si classifica inoltre in base alla dimensione dell'oggetto testato. Si ha:

- *testing in piccolo* se ci si riferisce alla verifica di un singolo componente (o modulo) di un sistema più complesso. Generalmente per questo tipo di verifica è necessario simulare anche l'ambiente col quale il modulo interagisce. Per far ciò si devono costruire almeno altri due moduli: uno *guida* che fornisce gli input ed uno *fittizio* che finge di essere stimolato dagli output del modulo testato.

- *testing in grande* se si verifica il sistema nella sua interezza, dopo aver fatto testing in piccolo delle sue componenti. Con questo tipo di testing il sistema complesso può essere assemblato seguendo un approccio *incrementale* (di tipo *top-down* o *bottom-up*) oppure *non incrementale* (anche detto *big bang test*, più caotico e rischioso).

Infine, per definire la gerarchia e l'ordinamento tra i test, sono usati i seguenti termini:

- *test case*: rappresenta l'unità logica tra i test. É pensato per verificare un singolo specifico comportamento o una caratteristica del software;

- *test script*: sono istruzioni che descrivono come uno o più test case devono essere eseguiti;

- *test suite*: è una collezione ordinata di test case e test script da eseguire per verificare un'area logica o fisica complessa di un software;

- *test plan*: è un documento di gestione che pianifica lo scheduling dei test, sottolineando i rischi e le priorità.

Scendendo in maggior dettaglio si potrebbero adesso elencare le varie classi di software testing (caricamento, sicurezza, regressione e così via), ma si preferisce rimandare la descrizione al capitolo tre, dedicato al web testing, dove queste tipologie si specializzano nell'ambito della rete.

1.3 Gli obiettivi del software testing

Praticamente ogni programmatore conosce la frustrazione di scoprire che il software da lui prodotto contiene ancora errori, nonostante le molte prove fatte. Anche quando si è praticamente certi del funzionamento di un software, in realtà non è possibile predire il suo comportamento in tutti i contesti in cui sarà utilizzato. Spesso non si può far altro che valutare i casi più ovvi e lasciare che i restanti errori vengano rilevati e comunicati dagli utenti, per poi essere corretti.

Quali sono le possibili cause di un errore? Perché è sfuggito ai controlli? Si possono elencare quattro casi generali che favoriscono l'insorgere di bug:

- l'utente ha eseguito una parte di codice mai testata;
- l'ordine in cui dichiarazioni ed espressioni sono eseguite non è mai stato provato in un test;
- l'utente ha introdotto una combinazione di valori in input che non era stata testata;
- l'utente si trova ad eseguire il software in un ambiente (sistema operativo, componenti hardware e così via) che non è stato considerato precedentemente.

L'attività di software testing non si pone come obiettivo la verifica di tutte le possibili situazioni critiche, quanto l'organizzazione e la selezione delle metodologie capaci di minimizzare il numero di errori non rilevati come pure il costo dei test. Nel far questo si individuano le funzionalità del software la cui affidabilità è indispensabile per l'utente, si valutano i rischi a cui ci si espone nel non correggere alcuni malfunzionamenti e si cerca di porre un equo limite al tempo necessario per testare un prodotto prima di immetterlo sul mercato.

Nella fase successiva, un ulteriore obiettivo sarà l'adeguata correzione degli errori rilevati, che eviti l'insorgere di nuovi errori o conflitti.

In generale l'obiettivo del software testing è quello di offrire una valutazione oggettiva della correttezza di un software, sulla quale gli sviluppatori stessi o terze parti possano basarsi.

1.4 Rilevare un bug

Certe volte, nell'incontrare un bug viene da chiedersi come è possibile che il programmatore che ha realizzato una certa procedura non se ne sia accorto prima. In realtà rilevare i bug (anche quelli "vistosi'') non è banale. Vi sono almeno due giustificazioni per il "reo'' programmatore:

- quella che per un utente può essere una sequenza ovvia e fondamentalmente necessaria di operazioni, agli occhi di un programmatore o di un verificatore (che non conosce tutti i possibili utilizzi del software) è una sequenza senza importanza. Può così aver deciso di escluderla dai test oppure l'esclusione può essere stata casuale ed involontaria. Questa

mancanza di cognizione pone l'accento sull'importanza della comunicazione tra utenti e sviluppatori durante tutto l'arco della produzione del software;

- le competenze per effettuare una buona attività di testing non sono per nulla banali. Linguaggi formali, fondamenti della teoria dei grafi e caratteristiche di algoritmi e strutture dati sono conoscenze necessarie per il testing ed al tempo stesso non così comuni tra gli sviluppatori. Anche per utilizzare tool di testing altrui serve una certa perizia, che si consegue solo dopo adeguati corsi di addestramento, molte volte mai realizzati.

1.4.1 Alcune tipiche complicazioni

Persino una semplice procedura di poche righe può nascondere insidie non visibili a colpo d'occhio, in grado di trarre in inganno anche gli analisti esperti. Si pensi ad una procedura per cambiare l'orario e la data di sistema, apparentemente semplice da testare: deve restituire l'orario e la data correnti e permettere all'utente di impostare valori aggiornati. Ecco tre aspetti che invece possono complicare la situazione:

- gli utenti che interagiscono con la procedura sono "sorprendentemente" due: uno è l'utente umano (esplicitamente) e l'altro è il sistema operativo (implicitamente). Non basta verificare le possibili interazioni tra procedura ed utente umano ma occorre anche supporre situazioni fuori dal normale per quel che riguarda le relazioni col sistema operativo. Il clock di sistema funziona bene? Manca la memoria necessaria? Se un altro programma cambia concorrentemente a sua volta l'orario? Cosa accade in questi casi?

- dato che il codice sorgente ha varie strutture di controllo del flusso, è possibile percorrere ogni ramo nel flusso del programma? Per un esempio come quello descritto in precedenza probabilmente sì: le possibili combinazioni *true/false* di ogni controllo sono in numero limitato e così è possibile costruire una *tabella di verità* per tutti i casi verificabili. Ciò nonostante i casi da valutare tendono a crescere esponenzialmente in proporzione al numero di condizioni true/false presenti nel codice. Se x è il numero dei controlli di flusso allora i casi possibili tendono a crescere come 2^x, per cui non è pensabile controllare tutti i casi per un x corrispondente a grandi software. Si deve ricorrere ad una selezione tra i percorsi del flusso. Ma secondo quale criterio?

- il dominio dei valori di input è troppo grande per essere testato per intero. Si consideri quante date, di formato legale ed illegale, può un utente cercare di impostare. Tipicamente si selezionano per il test input significativi o una loro sequenza logica. Nel caso della procedura di modifica della data si potrebbe scegliere di impostare come nuova data ed ora la mezzanotte del 1999 (il famoso "Millennium Bug''). Ma chi assicura un buon funzionamento per ogni possibile data inseribile?

Questi ed altri problemi di testing possono ulteriormente complicarsi nelle moderne applicazioni concorrenti, dove le variabili sono collegate logicamente tra loro e le sequenze di eventi verificabili hanno un ordine imprevedibile. Cambia inoltre la difficoltà nel fare un buon testing a seconda che i verificatori abbiano o meno conoscenza diretta del programma da testare (cioè abbiano la possibilità di effettuare sia testing black box che testing white box).

Una buona attività di testing per la rilevazione di bug, insomma, non si riduce solamente alla creazione ed all'utilizzo di appositi tool di verifica né all'applicazione meccanica di una data metodologia, ma deve comprendere anche tutta una serie di analisi propedeutiche piuttosto lunghe ed impegnative.

1.5 Le cinque fasi operative del software testing

Al fine di chiarire quali sono le principali problematiche da affrontare nel software testing ed al tempo stesso raggrupparle in classi omogenee da superare sequenzialmente ed in modo iterativo, si definiscono cinque fasi operative:

1. Modellazione dell'ambiente di lavoro.
2. Selezione degli *scenari* di test.
3. Esecuzione e valutazione degli scenari.
4. Correzione degli errori rilevati.
5. Misurazione dei progressi nel testing.

1.5.1 Modellazione dell'ambiente di lavoro

L'ambiente nel quale il software si colloca è forse l'aspetto più difficile da considerare nel suo insieme. Chi si appresta a testare un software non può

prescindere dal simulare anche il contesto nel quale il software opererà: è ricreando "artificialmente" (quanto meglio si può) le reali condizioni di lavoro del software che si può sperare di individuare i *fault* più comuni. La ricerca di un ambiente idoneo è però complicata da tutti i formati, protocolli ed interfacce diversi disponibili nell'informatica moderna. Si individuano quattro tipi di interfacce di ambiente da selezionare, combinare e testare:

- le *interfacce umane*, ovvero i possibili meccanismi di input di dati da parte dell'utente umano (dalle tastiere ai mouse, dalle *Graphical User Interface* ai *prompt* di comando);
- le *interfacce software*, comunemente chiamate *API* (Application Programming Interface) che sono un tramite tra il software ed il sistema operativo, le basi di dati, le librerie dinamiche e così via;
- le *interfacce col File System*, ovvero i formati con i quali il software deve comunicare col sistema operativo (tipicamente il formato dei file);
- le *interfacce di comunicazione*, che permettono l'accesso diretto alle svariate periferiche che possono essere in colloquio col software (simili interfacce sono rappresentate da *driver* e *controller*).

Definire di quali interfacce di ambiente il software farà uso corrisponde a determinare l'ambiente tipico in cui sarà testato. I test dovranno rispondere anche a domande riguardanti effetti prodotti dall'ambiente stesso: la cancellazione di un file condiviso o il *reboot* di una periferica durante un'operazione come influiscono sul comportamento del software testato?

1.5.2 Selezione degli scenari di test

Dato uno stato iniziale ed un ambiente di lavoro, per *scenario di test* si intende la configurazione globale degli input con il quale il software sarà eseguito.

La verifica di ogni singolo scenario di test ha un suo costo monetario e richiede del tempo. Si devono stabilire dunque dei criteri che permettano di selezionare gli scenari da testare in maniera intelligente. Solitamente i verificatori puntano sul cosiddetto *criterio di copertura*, dove per copertura ci si riferisce all'esecuzione di ogni linea di codice oppure all'applicazione di ogni evento ammissibile almeno una volta. In particolare per "esecuzione di linee" si intende l'attraversamento di cammini di esecuzione all'interno del codice, mentre "l'applicazione di eventi ammissibili" è in realtà una sequenza di eventi che simulano un utilizzo logico del software. Sfortunatamente,

sia i cammini che le sequenze nell'esecuzione di un software possono essere infinite. Ci si riconduce così al problema della decidibilità solo parziale della correttezza del codice e dunque la scelta del giusto insieme di scenari significativi rimane un compito arduo.

Altri approcci per la selezione degli scenari di test possono essere quelli in cui ci si affida al caso oppure a considerazioni statistiche e combinatorie.

Un obiettivo dei verificatori rimane comunque la definizione di un *test data adequacy criteria*, cioè di un criterio guida che individui l'insieme degli scenari di test appropriati (con input significativi) in relazione al software in oggetto. In genere la determinazione di un tale criterio è pensata per successivi test basati sulla copertura dei cammini o delle sequenze di eventi. Non si deve però dimenticare che il controllo del flusso è solo uno degli aspetti che possono guidare nella scelta degli scenari di test idonei: ad esempio, si possono scegliere anche scenari che mettono alla prova la coerenza di variabili e strutture dati o la complessità di esecuzione.

Un'importante caratteristica degli scenari di test, qualunque sia la finalità a cui sono rivolti, è la *ripetibilità*. É ovvio infatti che un test non ripetibile o non deterministico nei suoi risultati non ha molto valore.

1.5.3 Esecuzione e valutazione degli scenari

Una volta identificati gli scenari da testare in base al test data adequacy criteria il verificatore deve simularli sul software. É in questa fase che entrano in scena i tool di testing automatici che velocizzano e facilitano l'applicazione dei test, riducendo al tempo stesso le alte probabilità di errore di un analogo test condotto manualmente. All'utilizzo di tool per la verifica *run-time* (cioè in esecuzione) del software vanno di solito affiancati anche quelli per un'analisi statica del codice. Gran parte del lavoro è svolto in questo senso dai compilatori e dai *parser*, ma alcune volte è ancora richiesta la noiosa e certosina verifica manuale del codice riga per riga.

Completata la fase di esecuzione del test, si passa alla più difficile valutazione dei risultati ottenuti. In sostanza il verificatore deve controllare che il test abbia confermato un comportamento del software conforme ai requisiti richiesti. In alcuni casi è un passaggio semplice, perché si conoscono a priori i risultati validi oppure ci si può basare su test precedenti andati a buon fine. In altri casi però questo tipo di comparazione non è fattibile. Si pensi ad un software che restituisce numeri primi molto grandi: non è banale valutare se ogni numero è effettivamente primo o se c'è qualche numero primo che il software non è in grado di riconoscere.

A queste difficoltà di interpretazione deve solitamente far fronte un *ora-*

colo umano, specializzato nel dare una misura della bontà dei risultati del test.

Normalmente un oracolo si affida a due principali tipi di strumenti: i *formalismi descrittivi* ed il codice *embedded* (incastrato) nel codice sorgente, entrambi relativi alle specifiche software da testare. Sia modelli formali (grafi, diagrammi degli stati, grammatiche regolari) che informali (metalinguaggi o linguaggi naturali strutturati) sono molto utili per capire se il risultato ottenuto dal test è positivo o negativo perché definiscono chiaramente i requisiti ed il comportamento corretto del software. Numerosi formalismi più o meno astratti sono disponibili per i linguaggi procedurali come per quelli orientati agli oggetti, rendendo la descrizione delle caratteristiche di un software il più possibile non ambigua. É importante che un *oracolo* abbia a disposizione tali formalismi perché, a volte, caratteristiche proprie del software non "catalogate" in precedenza, possono essere considerate errori. Inoltre, tramite i formalismi è permessa un'analisi simbolica dei test, astraendo dalle complicate combinazioni di eventi che possono verificarsi nelle odierne applicazioni grafiche. Va però detto che, in alcuni casi, realizzare dei modelli completi è difficile se non proprio inutile, visto che le dimensioni finali del modello finiscono per superare quelle del software (si pensi al fenomeno dell'esplosione degli stati nei diagrammi a stati finiti).

La seconda metodologia di valutazione a disposizione degli oracoli è l'aggiunta di codice di verifica al codice sorgente vero e proprio. Di questa presenza l'utente non ha percezione. L'oracolo può invece far riferimento a questo codice per avere informazioni a tempo di esecuzione sullo stato interno del software, tramite apposite API o debugger. L'esempio più banale di questa tecnica di analisi sono i comunissimi "print'' provvisoriamente sparsi qua e là per restituire informazioni su oggetti e variabili, mentre meccanismi di test "built-in'' più evoluti sono le *asserzioni*.

Una tecnica meno usata che dovrebbe limitare l'impiego di oracoli umani per la valutazione dei test, è quella del *self-testing*. L'utilizzo di apposito codice embedded permette, ad altri software, di effettuare operazioni di comparazione automatica dei risultati del test con tabelle di valori noti e con risultati precedenti. Una variante sul tema è quella in cui si applica automaticamente un processo di *undo* e *redo*: se effettuando tutte le operazioni inverse a quelle svolte dal software testato si torna in uno stato identico a quello precedente il test, il risultato dovrebbe essere corretto. Anche i risultati del self-testing sono però da prendere con cautela perché non tutte le operazioni di test si adattano a questa tecnica. Inoltre, c'è sempre la possibilità che errori presenti sia sul software testato che su quello di self-testing si "mascherino" a vicenda.

In questa fase di esecuzione e valutazione degli scenari di test è quasi

sempre utile domandarsi se è il caso di rieseguire anche test precedenti, per controllare che una certa funzionalità sia ancora valida oppure che bug già rilevati siano stati veramente corretti. In questo senso il testing è una verifica della famosa *compatibilità all'indietro*. Questo modo di procedere implica l'esecuzione di appositi *test di regressione* (analizzati in maggior dettaglio nel capitolo 3), i quali hanno un ruolo molto importante nella verifica del software. Infatti la correzione di un bug può:

1. fissare il problema al quale era riferita;
2. rivelarsi insufficiente alla risoluzione del problema;
3. fissare il problema, ma crearne involontariamente di nuovi;
4. essere una combinazione dei punti 2 e 3.

D'altro canto è proibitivo eseguire ad ogni modifica un numero sempre più elevato di scenari di test precedenti. Così facendo si rubano tempo e risorse ai nuovi scenari volti a testare gli aggiornamenti delle ultime versioni del software. Oltretutto la riapplicazione di vecchi scenari può non coincidere con le priorità di test selezionate nell'attuale test data adequacy criteria. É quindi auspicabile la più stretta collaborazione possibile tra sviluppatori e verificatori al fine di garantire che le correzioni fatte siano adeguate e senza "effetti collaterali''. Solo così i test di regressione possono essere limitati nel numero e risultare al tempo stesso sufficientemente efficaci.

1.5.4 Correzione degli errori rilevati

Questa fase segue ovviamente la rilevazione dei bug ma non ha in realtà una collocazione precisa all'interno del processo di software testing. Una volta eseguiti dei test con successo (ovvero si sono individuati dei malfunzionamenti) occorre porvi rimedio, ma in base alla tipologia ed alla gravità dei bug si può decidere di intervenire in modi differenti. Ad esempio si potrebbe ritenere un bug così grave da dover riconsiderare completamente l'intera architettura del software, rendendo inutili i test seguenti. Al contrario si potrebbe valutare un bug sostanzialmente non dannoso e dunque rimandare la sua correzione a qualche fase seguente. Inoltre la presenza di α e β testing presuppone varie fasi nella correzione degli errori, che possono anche procedere parallelamente.

In sostanza la correzione dei bug deve rispettare gli esiti dei test ma può essere effettuata in vari momenti del software testing. É essenziale saper correggere i bug rilevati al momento opportuno: senza rallentare lo sviluppo

ma anche evitando di sprecare risorse in attività fuori luogo. Soprattutto, come già accennato nella sottosezione precedente, è essenziale che ogni correzione sia efficace e definitiva, per non costringere i verificatori a riconsiderare troppi scenari nei test di regressione.

1.5.5 Misurazione dei progressi nel testing

L'ultima fase del processo (iterativo) di testing è quella del monitoraggio dei risultati. "A che punto è il testing di questo software?''. É sempre difficile rispondere a queste domande perché la relazione tra i risultati quantitativi e qualitativi ed i progressi ottenuti non è banale. Il rilevamento di pochi errori può essere segno che il software è di qualità o all'opposto che i test non sono adeguati. Viceversa anche l'aver trovato molti errori gravi non esclude che ve ne siano ancora tanti altri. Di norma, in base al software da testare, si realizzano delle *check list* di obiettivi da raggiungere, lasciando ai verificatori la responsabilità di considerarli superati. Ad esempio per quanto riguarda la struttura di un software si potrebbero realizzare check list come la seguente:

- controllare i più comuni errori di programmazione;
- eseguire ogni percorso nel codice almeno una volta;
- inizializzare ed usare ogni struttura dati almeno una volta;
- controllare lo stato delle strutture dati durante l'esecuzione ed al termine del programma.

Un'analoga lista sotto il punto di vista funzionale potrebbe essere:

- applicare quanti più scenari di test possibile;
- esplorare i vari stati del software;
- eseguire sequenze di azioni che presumibilmente saranno effettuate dagli utenti.

Concludendo questo breve percorso attraverso le cinque fasi operative del software testing si può notare quanto sia importante la stretta collaborazione tra sviluppatori, verificatori ed utenti del software, durante tutto l'arco del processo di sviluppo. Progettare un software avendo già presenti le esigenze di test porta al concetto di *testability*, o capacità del futuro software di essere facilmente testabile. Programmare dei tool di verifica assieme alle singole procedure aiuta da una parte lo sviluppatore a comprendere meglio i requisiti

che il suo codice deve rispettare, dall'altra il verificatore a trovare degli strumenti ad hoc per le sue analisi. Qualità come la testability sono considerate essenziali in metodologie di programmazione relativamente recenti, qual'è, ad esempio, l'*eXtreme Programming* (XP).

1.6 eXtreme Programming: il testing come filosofia

Una delle metodologie di programmazione che si distingue per la capacità di riassumere alcuni aspetti tra i più produttivi ed efficaci nella produzione del software è l'eXtreme Programming (XP). L'XP si colloca tra le metodologie *leggere* (Lightweight o Agile methodologies), le quali predicano sia la flessibilità dei processi di sviluppo che la considerazione del fattore umano quale elemento centrale per lo sviluppo di buon software. Questa filosofia si contrappone alle metodologie pesanti tradizionali che all'opposto, costringendo tutte le fasi della produzione in schemi predefiniti, si distinguono per la rigidità nei cambiamenti delle specifiche e nella definizione dei ruoli. Non si elencheranno adesso tutti i molteplici aspetti dell'XP, ma ci si soffermerà sul software testing, che, all'interno dell'XP, ha un'importanza particolare. Infatti, nell'XP ogni fase della codifica deve essere affiancata da quella di testing per ritenersi completa. Le linee guida proposte sono essenzialmente quattro e ricalcano in gran parte le conclusioni raggiunte al termine delle sezioni precedenti:

1. *All code must have unit test*: ogni modulo di codice prodotto deve avere un relativo test per verificarlo. Questi test devono essere creati con appositi *unit test framework* (come JUnit) capaci di generare delle *suite di test* automatizzate. Nessun codice può essere integrato in un progetto senza che tali suite siano presenti e disponibili. Questa condotta consente a chiunque la verifica delle modifiche apportate e permette l'esecuzione dei test di regressione senza sforzi aggiuntivi;

2. *All code must pass all unit tests before it can relased*: il codice deve ovviamente passare tutti gli unit test presenti prima di essere rilasciato;

3. *When a bug is found tests are created*: quando un bug è rilevato devono essere subito creati ed inseriti nel progetto dei test particolari (*acceptance test*) in grado di evidenziarlo anche in futuro. Gli acceptance test sono in genere derivati dalle comunicazioni degli stessi utenti (*user stories*). La presenza degli acceptance test impedisce di aggiungere

codice al progetto nell'eventuale falsa convinzione di aver corretto un bug precedente ed al tempo stesso permette un rapporto più diretto tra utenti e sviluppatori;

4. *Acceptance test are run often and the score is published*: il cliente deve proporre, oltre alle user stories con le quali descrive come ha incontrato i bug, anche dei possibili scenari di test, che in questo caso sono le configurazioni di utilizzo per lui essenziali al fine di considerare corretti gli errori. A partire da questi scenari si creano gli acceptance test che devono risultare tutti positivi affinché una storia sia considerata "completata''. L'applicazione degli acceptance test dovrebbe essere ripetuta nel tempo e per quanto possibile automatizzata. I risultati dei test dovrebbero essere poi pubblicizzati sia tra gli sviluppatori che presso gli utenti.

Si potrebbe aggiungere molto su questa metodologia di programmazione che per alcuni sviluppatori è quasi una disciplina, ma in questo contesto ci si limita ad evidenziare la sua capacità di conciliare l'esigenza di software di qualità con le leggi della produzione.

1.7 ROC: se i bug sono considerati inevitabili

Il punto di vista dell'XP è sicuramente valido ed è in perfetta sintonia con quanto affermato nelle precedenti sezioni di questo capitolo. Ciò nonostante evitare quanto più possibile di produrre software errato non è l'unico approccio al problema ed in certi casi neanche la via migliore. Infatti, come si è dimostrato, la presenza di bug nel software è praticamente inevitabile e l'attività di testing, per quanto efficace, è uno strumento limitato. A queste constatazioni teoriche se ne possono aggiungere altre derivate dalla realtà: un recente sondaggio del settimanale americano Informationweek [9] tra 800 manager responsabili di altrettante aziende di business-technology ha messo in evidenza che:

- nel 2001 il 97% di loro ha avuto danni (ritardi, perdite di dati e così via) derivati da bug nel software utilizzato;
- i principali "colpevoli" sono compresi nelle tipologie di software maggiormente studiate ed usate: sistemi operativi, applicazioni da ufficio (word processor, fogli di calcolo,...), tool di sviluppo;
- secondo il 62% dei manager le aziende produttrici non stanno svolgendo una buona attività di testing per evitare l'insorgere di bug;

- solo il 5% si ritiene pienamente soddisfatto dal software commerciale di testing provato, percentuale che sale ad un misero 15% quando il software è realizzato appositamente per una specifica azienda.

Tutto ciò nonostante l'impegno nella produzione di software valido. Simili dati non confortano chi scommette sul testing, anzi, fanno apparire ancora di più la caccia ai bug una "lotta contro i mulini a vento". In realtà, quanto detto sull'utilità del testing rimane valido, ma alcuni stanno iniziando a considerare una via parallela nella ricerca di software bug-free. Alla Stanford University ed alla California University un gruppo di ricercatori coordinato da Armando Fox e David Patterson sta mettendo a punto una nuova metodologia di programmazione chiamata *ROC* (Recovery-Oriented Computing), [14] basata sul recupero veloce da situazioni di errore, partendo dal presupposto che gli errori saranno comunque presenti.

Attualmente la complessità dell'hardware e del software (specie quello di rete) cresce esponenzialmente e l'attività di testing non riesce a starle dietro. L'ottimizzazione del codice e la ricerca di bug poi interessa essenzialmente gli errori interni al software ma tende a trascurare la prima causa di errore, quella umana, dovuta essenzialmente a scarsa usabilità. Nel ROC l'obiettivo diventa la ricerca di un rapido (e "dolce") recupero delle funzionalità interrotte in seguito ad un bug. Infatti, per molte applicazioni di rete (e non solo), ridurre il tempo di inattività è più significativo che minimizzare la frequenza di errori, in modo da non dare all'utente percezione che qualche cosa stia andando storto.
Patterson e Fox indicano per il ROC quattro principi guida:

1. consentire un rapido recupero (riavvio) del singolo modulo (o insieme di moduli) in errore, evitando nella maggioranza dei casi di azzerare l'intero sistema o l'applicazione, con conseguente perdita di tempo e dati;

2. identificare chiaramente il bug che ha provocato un errore;

3. rendere disponibile per ogni operazione possibile una funzione di *undo* capace di riportare il sistema o l'applicazione ad uno stato antecedente l'insorgere del bug;

4. applicare in modo più organizzato e costruttivo l'analisi mutazionale, ovvero l'inserimento volontario di errori nel software, al fine di valutare l'efficacia dei test che dovrebbero individuare i bug ma anche di scoprire se il software li sa gestire come ci si attende.

In concreto il team di ricercatori sta cercando di raggiungere i suoi obiettivi attraverso un'adeguata programmazione modulare, tool automatici di monitoraggio dinamico e statistico, software di testing e *benchmarking* in grado di evidenziare come tali accorgimenti influiscano in modo sostanziale sui rendimenti di certe applicazioni.

Sicuramente, come implicitamente affermano gli stessi Fox e Patterson, il testing rimarrà un caposaldo del buon sviluppo software, ma non può risolvere il problema dei bug da solo (ed i dati lo dimostrano). Affiancargli metodologie come il ROC permette di considerare finalmente entrambe le facce della stessa medaglia: oltre a combattere i bug se ne alleviano anche gli effetti, con indubbio vantaggio per l'utente, gli operatori e la qualità del software in generale.

1.8 Conclusioni

Come si è tentato di dimostrare in questo excursus sul mondo del software testing, per quanto le tecniche siano complete ed elaborate, l'attività del verificatore continua ad avere un'anima "estrosa", fatta di intuizione ed esperienza, che va al di là di tante euristiche. É in ogni caso insospettabilmente complessa e destinata quindi a personale adeguatamente istruito su queste problematiche.

Capitolo 2

Applicazioni di rete e Web Service

Prima di passare al web testing viene affrontata in questo capitolo una breve descrizione delle applicazioni di rete, della loro struttura e delle loro principali caratteristiche. Il software pensato per le reti, ed in particolare per il Web, è basato su paradigmi particolari che si distaccano dal contesto del software tradizionale. Conoscere queste differenze aiuta a comprendere dove e come il web testing si specializza rispetto al software testing.

2.1 Il paradigma Client/Server

Il paradigma client/server, nella maggioranza delle applicazioni di rete, definisce le modalità di scambio delle informazioni tra soggetti distinti, permettendo di fatto la realizzazione dei servizi.

Internet ed in generale le reti costituiscono soltanto un'infrastruttura che permette di comunicare. Allo stesso modo l'hardware di rete ed i relativi protocolli non sono altro che mezzi, i quali non prendono nessuna iniziativa nel dare inizio ad una nuova comunicazione. Sono invece le applicazioni i soggetti che, ad un alto livello di astrazione, sanno quando, come e con chi iniziare una conversazione.

Un'importante differenza tra una normale telefonata ed una comunicazione in rete è che nella seconda le applicazioni non hanno una diretta percezione della controparte. Esse devono appoggiarsi a determinati protocolli che le informino quando arriva un messaggio e da parte di chi, se è della forma attesa oppure se è stato ricevuto. I soggetti che dialogano, per evitare confusio-

ne, sono costretti ad osservare una rigida organizzazione delle informazioni scambiate, assumendo in modo coordinato un ruolo attivo o uno passivo nella conversazione.

Nel paradigma client/server questi soggetti hanno un ruolo più definito: l'applicazione che prende l'iniziativa del contatto viene chiamata *client*, mentre quella che lo attende è il *server*. Non bisogna fraintendere i significati dei termini client e server: spesso nel gergo informatico si indicano impropriamente con gli stessi nomi le macchine sulle quali tali applicazioni sono eseguite. In realtà su un singolo computer adibito a "server" possono operare più applicazioni server contemporaneamente, per cui è consigliato associare a tali macchine dedicate il nome di "Server-class computer" e "Client-class computer". In generale un client

- è un generico programma applicativo, utilizzabile per computazioni locali, che diventa client quando è richiesta una comunicazione remota;
- è invocato da un utente e la sua durata non coincide necessariamente con la vita del sistema nel quale risiede;
- dispone di un'interfaccia utente;
- può contattare più server per volta.

Un server invece

- è un programma specializzato nel fornire un certo servizio;
- può gestire più client contemporaneamente;
- vive durante ogni fase della vita del sistema sul quale risiede;
- richiede hardware e sistemi operativi appropriati;
- generalmente non dispone di un'interfaccia utente.

Il modello client/server impone l'asimmetria nella comunicazione, collocando le applicazioni su un piano non paritetico (al contrario di quanto accade nel modello *peer-to-peer*). La conversazione può svolgersi sia a senso unico sia in entrambe le direzioni. Ad ogni richiesta del client seguono una o più risposte del server e solo in alcuni casi il server risponde con un output continuo (ad esempio nel caso di uno stream video).

Ma come fanno il client ed il server a scambiarsi i dati? E come riescono a contattarsi? Per risolvere simili questioni si fa affidamento sui vari protocolli che agiscono a livelli di astrazione diversi. Nell'ambito Web una delle coppie

di protocolli più usata è *TCP/IP*: il primo (Transmission Control Protocol, protocollo di trasporto) permette di stabilire un canale di comunicazione tra un'applicazione ed un'altra; il secondo (Internet Protocol, protocollo di collegamento), ad un livello più basso, consente di indirizzare e spedire le informazioni. Tramite questi protocolli è possibile per il client specificare quale servizio richiedere presso un dato server e per il server rispondere in modo diretto al client richiedente. Senza scendere in maggiori dettagli si può schematizzare il dialogo tra client e server come segue:

1. si assegna ad ogni servizio offerto un identificatore univoco;
2. un server viene avviato e si registra presso il sistema rendendosi disponibile a soddisfare un servizio con un determinato identificatore;
3. il server resta in attesa di chiamate;
4. il client chiede al proprio software di protocollo quali sono gli identificatori di servizio presso una data macchina dall'indirizzo noto;
5. ottenuto l'identificatore del servizio voluto, il client apre un canale di comunicazione con il server resosi disponibile a soddisfarlo;
6. all'arrivo della richiesta il server sospende l'ascolto di altre chiamate (se è sincrono) e comincia ad interagire col singolo client; oppure può gestire concorrentemente molteplici chiamate (se è asincrono) generando altrettanti sottoprocessi;
7. terminata la comunicazione, il server torna in attesa mentre il client chiude il canale di comunicazione precedentemente aperto.

A seconda che la funzione computazionale sia prevalentemente a carico dell'una o dell'altra parte si parla di *fat client* o *fat server*. Quella dello sbilanciamento computazionale è una delle caratteristiche da valutare con maggior attenzione: attualmente, vista la notevole potenza di calcolo raggiunta dai singoli pc e l'aumento vertiginoso di connessioni ad Internet, la tendenza è quella del fat client, così da evitare, quando possibile, un superlavoro ai server. La funzionalità "lato-server'' resta indispensabile per certe operazioni, quali, ad esempio, l'accesso a *database* centralizzati. Un tipo più sofisticato di server è infine quello *multi-livello*, il quale, per soddisfare un certo servizio, è in grado di collaborare con altri server.

2.2 Il Web

Tra le varie applicazioni che hanno fatto la fortuna di Internet troviamo l'FTP (File Transfer Protocol), l'e-mail, il Network Video, Telnet, ma soprattutto il WWW (World Wide Web). Quest'ultima applicazione di rete, sviluppatasi nei primi anni '90, ha avuto un tale successo presso il grande pubblico da divenire sinonimo di Internet. La stragrande maggioranza delle applicazioni che oggi si trovano su Internet è pensata per il Web. In realtà WWW può essere visto come un insieme di client e server che dialogano usando il protocollo standard *HTTP* (Hypertext Transfer Protocol), il quale permette di trattare in modo uniforme quasi tutte le componenti raggiungibili tramite Internet. Le applicazioni progettate per il Web devono necessariamente comprendere questo protocollo oltre al linguaggio utilizzato per rappresentare le risorse sotto forma di ipertesto: *HTML* (Hypertext Markup Language).

L'accesso al Web avviene tramite dei particolari client chiamati *browser web*. Essendo solitamente completi di interfaccia grafica, i browser sono relativamente semplici da usare e per questo l'accesso alle risorse di rete tramite Web è divenuto possibile anche ad utenti inesperti. I browser moderni includono anche molti *plug-in*, moduli integrati capaci di rendere il browser adatto ad operazioni più complesse della semplice lettura di una pagina HTML: transazioni sicure, esecuzioni di *script* e *applet* Java, interpretazione di particolari formati multimediali sono solo alcuni esempi.

Le tipologie di server dedicate al Web sono più numerose, benché spesso non altrettanto distinte: i più importanti sono sicuramente i *web server* (contenenti le pagine HTML); poi vengono i *database server* (che fungono da deposito dati) e gli *application server* (che estendono i servizi offerti tramite linguaggi particolari), oltre ai più specifici *proxy server*, *search server* ed *e-commerce server*.

Ogni risorsa accessibile nel Web è dotata di un *URL* (Universal Resource Locator) univoco, al quale viene di solito associato un nome simbolico. L'associazione risorsa-URL-nome simbolico rappresenta un *link*.

Il Web è sostanzialmente un'applicazione di Internet capace di sommare varie funzionalità di rete e di collegare tra loro con la stessa facilità elementi multimediali, testi, database e così via. Tutta una gamma di *siti web*, dall'aspetto e dalle funzionalità più disparate, è oggi disponibile sul Web. Benché le applicazioni pensate per il Web non costituiscano la totalità del software di rete, ne rappresentano sicuramente la porzione più ampia ed appariscente. Praticamente ogni azienda ha un proprio sito web e moltissime sono ormai quelle che sul Web fondano i propri affari (le stesse che hanno dato vita al fenomeno "New Economy") fornendo particolari servizi. D'altro canto an-

che il software tradizionale si trova spesso a dover interagire col Web (per scaricare aggiornamenti, inviare e-mail, compilare moduli, ...).

Nel capitolo 1 sono state illustrate le ragioni del software testing. Viste le problematiche intrinseche delle reti (in questa tesi non affrontate) e la varietà di aspetti inediti presenti nel software progettato per il Web, il testing diviene ancor più necessario.

2.3 I servizi di rete

In che consistono i famosi servizi che i server possono soddisfare? Nell'immaginario informatico comune una rete (nell'accezione più generale) è un "deposito" in cui vengono messe a disposizione risorse ed informazioni di vario tipo. In Internet, la rete per eccellenza, si può trovare, tra le altre cose, il sito web di un comune, il servizio prenotazioni di un'agenzia di viaggi, l'e-banking di un istituto di credito, l'elenco on-line dei libri di un editore, ma anche una casella di posta elettronica o una chat. In generale, ogni volta che si raggiunge una risorsa tramite una rete, si accede ad un *servizio* che essa offre. Un servizio banale è quello di mostrare il contenuto di una pagina HTML; uno più complicato può essere una *chiamata a procedura remota* (Remote Procedure Call). Dietro ad ogni servizio stanno uno o più server, una o più applicazioni di rete, protocolli di comunicazione adeguati e hardware dedicato.

I servizi di rete sono già moltissimi ma ne vengono creati sempre di nuovi per sfruttare le nuove tecnologie e far fronte alle esigenze degli utenti. Alcuni servizi sono destinati alle reti interne di aziende ed organizzazioni, altri pensati per essere accessibili da qualsiasi pc dotato di *modem*. Si ha dunque un proliferare di applicazioni di rete con funzionalità, prestazioni, struttura, finalità e requisiti di compatibilità molto diversi tra loro. Se da una parte questo è positivo perché offre all'utente una maggiore libertà di scelta, dall'altra obbliga gli sviluppatori a considerare tutta una gamma di problematiche assente nel contesto del normale software centralizzato. L'interoperabilità delle componenti, la scelta dei linguaggi di sviluppo, la coordinazione e la collaborazione delle applicazioni, i limiti stessi delle reti (raggiungibilità, latenza e così via) complicano ogni fase dello sviluppo di un servizio di rete, ampliando la schiera dei possibili bug e dei relativi test da effettuare.

In particolare è oggi molto sentito il problema delle incompatibilità hardware e software. Da quando Internet ha permesso di collegare insieme decine di milioni di utenti, poter reperire un particolare servizio ed utilizzarlo a prescindere dalle piattaforme hardware e software è una delle principali sfide degli informatici moderni. Il testing stesso finisce per incontrare un grosso

limite nelle incompatibilità che costringono le applicazioni e le risorse di rete in un ambiente chiuso nei riguardi di piattaforme diverse. Le spese di cui si devono far carico le aziende produttrici di software per superare queste barriere e controllare poi che effettivamente siano state superate, sono molto alte. Qualcuno ha descritto metaforicamente Internet come un gigantesco *arcipelago* di isole-servizi, ognuna con un proprio dialetto. Non è possibile ipotizzare in questa situazione una *somma di servizi* su larga scala perché non avrebbero capacità di integrarsi agevolmente e vicendevolmente. Evidentemente questa non è una situazione favorevole allo sviluppo di determinati settori, come ad esempio quello dell'*e-business*. Per questo le grosse industrie dell'*Information Technology* si stanno accordando (e combattendo) per definire protocolli, linguaggi e piattaforme comuni.

É da questo sforzo che nasce il concetto di web service come estensione del servizio tradizionale.

2.4 I web service

Un *web service* può essere definito come *un'interfaccia che attraverso una rete descrive una collezione di operazioni accessibili mediante messaggistica XML.*

La caratteristica del web service rispetto al servizio web tradizionale è l'interoperabilità e l'indipendenza dalla piattaforma su cui il servizio risiede ed è stato codificato. Una rete di grandi dimensioni è infatti tipicamente formata da infrastrutture hardware e software spesso eterogenee. Con l'uso di strumenti quali XML, SOAP, UDDI, WSDL, si scavalcano le incompatibilità, stabilendo uno standard comune di messaggistica ad alto livello per trasferire informazioni provenienti da fonti completamente diverse. Così facendo si mettono in comune non soltanto i dati ma anche molte funzionalità dei servizi, rendendole fruibili da qualunque piattaforma che ne faccia idonea richiesta. É una trovata non originale nel mondo dell'informatica, ma rimane fondamentalmente valida se si vuol realizzare un sistema di *computazione distribuita* su grande scala (tramite Internet). Si pensi alla possibilità di creare dei macro-servizi da combinazioni di servizi semplici sparsi nella rete. O alla facilità con cui potrebbe essere scritta un'applicazione web se si sapesse che tutto ciò che viene ricevuto ha una codifica indipendente dalla fonte. Tutto questo si ottiene dotando il "vecchio servizio" di un'efficace astrazione. Con i web service si implementa di fatto *un'astrazione di servizio*, trasparente ai mezzi col quale il servizio è richiesto e svolto.

2.4.1 Architettura dei web service

I web service possono essere visti come dei *framework* di messaggi. Ad un web service è richiesto semplicemente di saper ricevere e spedire messaggi in accordo con i protocolli di rete. Nella maggior parte dei casi il loro funzionamento si riassume con uno schema classico:

- il server riceve dal client un messaggio interpretabile, al quale segue l'avvio di una o più applicazioni in remoto;
- terminata l'elaborazione viene inviato in risposta al client un messaggio con il risultato.

La novità sta nell'utilizzo di meccanismi standardizzati di impacchettamento dei dati, abbinati ad XML (eXtensible Markup Language), un linguaggio di codifica delle informazioni universalmente riconosciuto e sempre più utilizzato.

A livello logico un web service può essere scomposto in due componenti principali:

- un *service listener*, in grado di comprendere i protocolli di trasporto ed impacchettamento. Esso rimane in attesa di messaggi in arrivo;
- un *service proxy* in grado di tradurre le richieste in effettive chiamate all'applicazione retrostante.

Un'altra caratteristica dei web service è l'*integrazione dinamica* con la piattaforma sulla quale risiede l'applicazione server. Si vuole infatti garantire l'indipendenza da tale piattaforma ed al tempo stesso fornire un meccanismo che permetta di orientarsi nella ricerca di un servizio, con la libertà di variare i parametri dei servizi offerti. Questa configurazione "sul momento" è realizzata nella *web service architecture* da vari soggetti:

1. un *service provider* pubblica un elenco dei servizi che può offrire in un certo momento;
2. questo elenco è reso noto ad un *service registry* (fase di pubblicazione del servizio);
3. un *service requestor* o *consumer* (essere umano o altra applicazione web) si rivolge al service registry per trovare un servizio a lui utile e questo gli riporta l'indirizzo del service provider in grado di fornirglielo;
4. ottenuto tale indirizzo il service requestor tenta di accedere al servizio vero e proprio presso il service provider (fase di *bind*);

5. conclusa positivamente la fase di bind lo scambio dei dati avviene direttamente tra il service requestor ed il service provider.

Questa architettura permette di variare il numero, il tipo e l'indirizzo di certi servizi offerti anche a run-time, dinamicamente e senza coinvolgere necessariamente chi richiede il servizio.

Alternativo e complementare al modello di architettura analizzato c'è il *modello Peer*, che ha una configurazione più flessibile in quanto la distinzione tra service provider, registry e consumer non è rigida. Ogni peer (punto o nodo) della rete può costituire un'entità "polimorfa'' che assume a seconda delle necessità, dei servizi richiesti e del soggetto che li richiede, uno dei ruoli necessari alla realizzazione del servizio.

2.5 Conclusioni

In questo capitolo è stata data una veloce descrizione dei meccanismi che stanno dietro al software a cui si applicheranno le tecniche di web testing illustrate nel prossimo capitolo. Chi effettua web testing è però tenuto ad approfondire nei dettagli la propria conoscenza delle reti, delle loro qualità e problematiche. É fondamentale aver presenti i limiti delle applicazioni web e le loro diversità rispetto al software tradizionale perché sottovalutare o ignorare queste differenze può portare a test formalmente corretti ed efficaci nella teoria ma insufficienti nella realtà.

Capitolo 3

Il web testing

In questo capitolo saranno analizzati in modo più dettagliato i punti critici delle applicazioni web e di conseguenza dove si rende necessario il web testing. Non verranno proposti dei test specifici ma ci si limiterà ad evidenziare il loro obiettivo finale. Questo perché ogni team di sviluppo o di verifica ha a disposizione vari strumenti e, per effettuare test concreti, può combinarli con altrettante metodologie. Salendo invece ad un livello appena superiore è conveniente focalizzare l'attenzione sugli obiettivi del web testing, in particolare su come scomporre qualitativamente in categorie questa attività al fine di raggiungere più facilmente gli scopi preposti.

3.1 Cosa si deve testare

Dal software testing deriva (storicamente e metodologicamente) il web testing, per cui la maggior parte delle conoscenze accumulate per il primo è stata utile per il secondo. La diversità di contesto tra i due tipi di testing implica però differenti modalità di applicazione e valutazione. La verifica dei link, ad esempio, è un'operazione che non viene effettuata di norma per il software tradizionale. In ogni caso, anche quando gli aspetti da testare sono presenti sia nel web che nel software testing, è difficile che la semantica rimanga la stessa: uno script può essere visto come un metodo di un oggetto, ma quando si testa nel suo contesto molte somiglianze spariscono.

Un software che lavora localmente non viene spesso a contatto con problematiche quali lunghi tempi di download, latenze di rete, traffico ed indirizzamento delle informazioni, utilizzo condiviso da parte di vari utenti (anche numerosi). Altri aspetti come la sicurezza o l'autenticazione degli utenti hanno poi un peso minore. Persino l'interfaccia grafica, che per un

software tradizionale è definita una volta per tutte in fase di sviluppo, nelle applicazioni di rete può essere soggetta ad interpretazioni diverse da parte dei browser che la visualizzano.

Le caratteristiche tecniche non sono d'altra parte le uniche a fare la differenza. Nel momento in cui un sito web viene messo in rete, lo si espone all'utilizzo di un'utenza composita e variegata, di cultura, conoscenze informatiche ed interessi anche molto diversi. Questo avviene indipendentemente dalle reali intenzioni ed ambizioni degli sviluppatori. In altre parole l'usabilità, già fattore critico nel software testing, è ancora più importante nelle applicazioni web che interagiscono direttamente con utenti spesso non definiti in fase di progetto.

Per rimarcare ulteriormente quanto il differente contesto influisca sui test destinati ad un'applicazione web, si ricordi che quasi tutte queste applicazioni hanno la necessità di un aggiornamento continuo (a volte automatico) dei propri contenuti o della propria interfaccia. Quindi, mentre nel software "statico'' tutto (a parte il valore dei dati da immettere) è definito in fase di sviluppo, nelle applicazioni web si ha molta più "dinamicità'', a volte in funzione dell'utente stesso. Tutto questo fa sorgere numerosi problemi di coerenza, rendendo necessari altrettanti test.

La qualità di un'applicazione web può dipendere da un dato insieme di aspetti, tra cui:

- le interfacce utente;
- le funzionalità;
- l'accesso ai database;
- la facilità e la correttezza del processo di installazione/disinstallazione;
- la facilità di configurazione e la compatibilità;
- la sicurezza e l'autenticazione;
- le performance (con l'esecuzione in stato di stress).

Si noti che ognuno degli aspetti elencati (che non saranno necessariamente tutti presenti) è facilmente riscontrabile dagli utenti, dagli operatori o dall'amministratore. Per questo un bug che interessi una di queste caratteristiche può incidere notevolmente sull'affidabilità e l'usabilità dell'applicazione.

Oltre ai test riguardanti le peculiarità evidenziate, in fase di sviluppo rimangono da verificare alcune qualità specifiche meno appariscenti: l'architettura dell'applicazione, le modalità di dialogo tra le componenti, l'efficienza degli algoritmi ed in generale tutto quello che si addice ad un testing white box.

3.2 Una panoramica generale

I *test types* sono categorie di test pensate per catalogare determinate classi di errori. La suddivisione dei test in tipologie consente di individuare più agevolmente le aree logiche del software ed i test destinati a verificarle in determinate fasi dello sviluppo. I test sottoelencati sono comuni sia al software che al web test. Nella prossima sezione saranno riprese in esame dal più specifico punto di vista del Web.

3.2.1 Acceptance testing

- *Development acceptance test*: consiste in una serie di test effettuati in fase di sviluppo con l'obiettivo di assicurare un livello minimo di qualità. A sua volta si suddivide in

  1. *Release acceptance test (RAT)*: test effettuato per ogni *release* per verificare che sia sufficientemente stabile prima di effettuare ulteriori test. Normalmente consiste in controlli sui valori di input e di output.
  2. *Functional acceptance simple test (FAST)*: è il primo test funzionale effettuato su ogni release in fase di sviluppo, volto a controllare la presenza delle caratteristiche chiave nell'applicazione. Superare anche questo tipo di test è una condizione necessaria per poter successivamente applicare test più complessi.

- *Deployment acceptance test*: prevede l'installazione e la configurazione dell'applicazione in quello che sarà il reale ambiente di lavoro (o una sua versione approssimata). Spesso infatti, fino ad un certo grado di sviluppo, il software viene testato in ambienti di lavoro estranei a quello di destinazione. Collocare il programma nel contesto giusto è essenziale prima di procedere con test di livello superiore.

3.2.2 Feature-level testing

Accertata una correttezza di base del software si procede col verificare, stavolta in modo approfondito, le sue caratteristiche:

- *Task-oriented functional test (TOFT)*: i TOFT consistono in un'analisi positiva dei compiti che l'applicazione dovrebbe saper svolgere, sulla base di quanto stabilito nei documenti di analisi e progetto.

- *Forced-error test (FET)*: a differenza dei TOFT, i FET agiscono negativamente, costringendo l'applicazione a lavorare in condizioni di errore. Si sfruttano questi test per generare contemporaneamente una lista di errori possibili e verificare come il programma li gestisce.

- *Boundary test*: controllano come il programma risponde nei casi in cui l'input assume valori estremi o critici.

- *System-level test*: sotto questo nome vanno tutti i test volti a verificare come l'applicazione interagisce col sistema sottostante.

- *Real-world user-level test*: vengono eseguite le operazioni che si suppone siano più frequenti da parte degli utenti.

- *Load/Volume test*: questi test riguardano lo studio di come il programma gestisce grandi quantità di dati e computazioni eccessive (non necessariamente estreme).

- *Stress test*: in questo caso si forza il software a lavorare in condizioni di risorse limitate (memoria, spazio su disco, larghezza di banda sulla rete e così via). L'obiettivo è individuare il limite oltre il quale le funzionalità non sono più garantite.

- *Performance test*: l'obiettivo di questi test è scoprire la strategia giusta per mantenere le funzionalità del programma sopra un certo livello di efficienza nella maggior parte dei casi.

- *Regression test*: sono test usati per confermare che i vecchi bug siano stati corretti adeguatamente, senza danneggiare altre funzionalità. Sono di solito ripetuti ad intervalli predefiniti.

- *Compatibility and configuration test*: in questi test si vanno a controllare le funzionalità del software nei riguardi di particolari componenti esterne (periferiche, sistemi operativi, ...). Una strategia spesso adottata è quella che prevede l'esecuzione di un sottoinsieme di FAST o TOFT in vari ambienti di lavoro.

- *Documentation/On-line help test*: anche come il programma viene in aiuto dell'utente è un aspetto da testare. Viene controllata dunque l'accuratezza della documentazione, insieme all'usabilità ed alla funzionalità degli help in linea.

- *Install/Uninstall test*: testare che l'installer metta l'applicazione in condizioni di funzionare correttamente e l'uninstaller la rimuova dal

sistema, è una questione di "bon-ton'' informatico. L'utente non deve preoccuparsi di questi "dettagli" più di tanto: esso è interessato solo all'uso del software. Per le applicazioni web si può avere un'installazione/disinstallazione dal lato client, dal lato server o da entrambi i lati.

- *User interface test*: la facilità d'uso dell'interfaccia utente è un altro punto da testare. Con questi test si mettono alla prova l'usabilità, l'aspetto grafico, la navigabilità, l'accessibilità ed il *feedback* dato dell'interfaccia.

- *Security test*: per molte applicazioni la sicurezza è un aspetto cruciale. Con questi test si valuta se la politica adottata per rendere il software sicuro è sufficiente rispetto al livello di protezione richiesto.

- *Unit test*: rientrano in questa categoria di test tutti quelli volti a valutare la correttezza delle singole unit di codice prima che vengano integrate nel software. É un tipo di testing effettuato "in privato" dai singoli programmatori.

3.3 Analisi di alcune categorie di web testing

Segue ora la presentazione di alcune tra le principali classi di web testing attraverso le problematiche specifiche delle applicazioni web. In particolare verranno trattate quelle che più direttamente interessano il tool di web testing realizzato per la tesi (Bellerofonte) e l'applicazione web alla quale sarà applicato (Orbilio). Per maggiori dettagli si rimanda ai riferimenti bibliografici [11].

3.3.1 User Interface test

Testare un'interfaccia vuol dire verificare disegno ed implementazione delle sue componenti. Il testing delle UI (User Interface) è condotto sia con test appositi sia nel corso di altri test (ad esempio i TOFT o i test di usabilità).

- Per quanto riguarda il **disegno di interfacce**, le migliori valutazioni provengono dagli utenti finali. Naturalmente esistono anche delle autorevoli linee guida da seguire, basate su euristiche e considerazioni derivate dalla psicologia cognitiva. É però sempre utile coinvolgere campioni di utenza per avere impressioni dirette sull'usabilità, la navigabilità, la chiarezza e l'accessibilità dell'UI. Una buona interfaccia

deve in sostanza supportare l'utente in ogni possibile interazione con l'applicazione, ma deve farlo senza che l'utente si debba mai porre troppe domande.

É essenziale rispondere subito a due interrogativi:

- "Chi saranno gli utenti dell'applicazione?''. Anzitutto è necessario sapere se gli utenti dell'applicazione web si troveranno dal lato server o da quello client. Ai primi si attribuisce in genere una funzione amministrativa e dunque si suppone che il loro livello tecnico sia sufficientemente alto. Altrettanto non si può ipotizzare per gli utenti client-side: essi accederanno all'applicazione prevalentemente tramite un browser per usufruire del servizio offerto, ma spesso senza disporre di conoscenze specifiche. In questo caso l'interfaccia assume un ruolo-guida molto più rilevante.

  Per classificare l'utenza inoltre si ricorre ad un insieme di parametri qualitativi: esperienza informatica, esperienza del Web, conoscenza del campo applicativo del software ed esperienza nell'uso di applicazioni simili.
- "Quale approccio generale seguire nel disegno?''. Individuata l'utenza tipica, bisogna pensare a quale schema generale di disegno sia più adatto. Ad esempio, per supportare utenti "inesperti" si può considerare l'inserimento di numerose schermate di *wizard*, che guidino passo dopo passo attraverso scelte complicate. Importante è poi la *metafora* che deve accomunare ogni elemento del disegno: una metafora è un ponte tra le esperienze dell'utente nel mondo reale (ed informatico) e quelle fatte nell'applicazione. Ad esempio si può scegliere di rappresentare graficamente una funzione "calcolatrice" in tanti modi diversi, ma disegnarla simile alle calcolatrici reali aiuta sicuramente l'utente a capirne l'uso. Una metafora sbagliata disorienta notevolmente un utente.

Un programmatore non può prevedere tutte le singole impressioni che la sua interfaccia susciterà negli utenti, ma può verificare oggettivamente che il significato di un elemento sia consistente ogni volta che appare nell'interfaccia. Gli errori stessi dovrebbero essere presentati in modo coerente al resto del disegno.

Il disegno di un'interfaccia web non si limita ad interessare la "piacevolezza" dell'utilizzo, riguarda da vicino anche l'interazione in termini di input/output. Durante l'uso dell'applicazione, l'utente dovrà inserire spesso dei valori in input per poi attendersi la presentazione dei relativi output. In queste fasi l'applicazione lo dovrà guidare ed aiutare

tramite tutta una serie ben combinata di *controlli grafici*: dai semplici *tag* HTML (form, bottoni, ...) ai controlli dinamici realizzati in Java, ActiveX, JavaScript e così via, fino all'utilizzo di *Server-Side Includes* (SSI) e *Cascading Style Sheets* (CSS). La presentazione degli output è vitale quando si ha a che fare con dati provenienti da database in quanto occorre saperli organizzare in tabelle che ne permettano una facile e veloce consultazione.

- L'**implementazione dell'interfaccia** ha invece un occhio di riguardo per le operazioni che stanno dietro ogni elemento. In sostanza si richiede ad ogni componente di "fare quello che ci si immagina che faccia a giudicare dal disegno". I test di questo tipo sono spesso svolti in concomitanza di quelli funzionali. Alcune complicazioni specifiche del Web circa l'implementazione di interfacce sono:
  - la diversa interpretazione da parte dei vari browser;
  - l'invio ritardato al server degli input immessi dall'utente: fin quando l'utente non dà esplicito avvio alla trasmissione, i valori immessi sono presenti solo sul client e quindi c'è pericolo di perdita dei dati, benché l'interfaccia dia l'impressione che ogni inserimento sia definitivo;
  - l'esecuzione di script non è sempre consentita sul client;
  - il bottone "Back'', presente in ogni browser, complica le relazioni tra le pagine web, in quanto, consentendo di tornare all'ultima pagina visitata, può non rispettare i link presenti sulla pagina attuale;
  - la risoluzione di schermo ed i caratteri installati nel sistema influiscono sulla presentazione dell'interfaccia.

3.3.2 Functional test

Il testing delle funzionalità di un'applicazione web riguarda la verifica di ciò che l'applicazione dovrebbe fare, relativamente alle aspettative dell'utente. In questa categoria rientrano molte sotto-tipologie di test: i FAST, i TOFT, i boundary test, i forced-error test e parte dei security test.

- FAST: come descritto nella sezione precedente, i FAST mirano a dimostrare la correttezza delle funzionalità ad un basso livello, senza prendere in considerazione le combinazioni con altre funzionalità (testate nei

TOFT). Nel caso delle applicazioni web, uno dei loro obiettivi è verificare il comportamento dei singoli componenti dell'interfaccia (text-box, bottoni, bandierine, ma anche i link grafici e testuali), i loro eventuali valori di default e via dicendo. Sono considerate caratteristiche chiave in ambito Web anche operazioni come il log in/log out, la ricerca, l'autenticazione ed il recupero password smarrite.

- TOFT: nei TOFT si pone l'attenzione sulla capacità dell'applicazione di soddisfare compiti più articolati, esplicitamente richiesti nel progetto. In generale vengono eseguiti dopo i FAST, in quanto si basano su liste di caratteristiche di base prese già singolarmente in esame. É da stabilire in questo contesto se combinazioni opportune di queste caratteristiche riescono a soddisfare dei requisiti particolari. Ad esempio potrebbe essere richiesto che il task composto dalla sequenza log in e autenticazione venga eseguito entro un dato tempo.

- Forced-error test: lo scopo di questi test è trovare le condizioni di errore ignorate o gestite male. Ad esempio, se un campo di un modulo ammette come valide solo sequenze di caratteri alfabetici, l'introduzione di un numero genera una condizione di errore che deve essere gestita correttamente affinché il test sia considerato superato. Per ogni condizione di validità ce n'è sempre una di invalidità. La maggiore complicazione nell'ambito Web consiste nella quantità di soggetti che entrano in gioco per alcune transazioni: oltre al client ed al server vi è tutta una catena di "intermediari" (tra cui la rete stessa) che concorrono alla realizzazione di un certo servizio, ognuno dei quali può generare errori difficilmente prevedibili (e gestibili) nel complesso.

- Boundary test: i boundary test considerano i limiti nel dominio di ogni funzionalità e la mettono alla prova per tali valori. Possono considerarsi delle estensioni dei forced-error test e dei TOFT.

Una sottotipologia che riassume un pò tutte quelle elencate è quella degli *Exploratory test*, che consiste nello "spostarsi'' all'interno dell'applicazione e testare di volta in volta le funzionalità incontrate.

3.3.3 Database test

Tutte le applicazioni web che necessitano di accesso ai dati hanno bisogno di un *database server*. I database giocano un ruolo importantissimo nel Web. Essi ospitano i dati delle applicazioni e gestiscono le loro operazioni di inserimento, cancellazione ed interrogazione. Una delle tecnologie comunemente

usate per i web-database è quella dei *database relazionali*. I database relazionali sono composti da tabelle che possono essere facilmente riorganizzate e visitate. I dati sono contenuti in *record* relativi a dei campi a loro volta appartenenti alle tabelle. In ambito Web poi, l'archivio è spesso disperso tra molteplici server, per questo si parla di *database distribuito*. Il database server viene normalmente implementato tramite un apposito linguaggio, *SQL* (Structured Query Language), che offre tutta una serie di primitive per manipolare i dati nelle tabelle bidimensionali.

Anche nel caso dei database vi sono molti punti di interazione tra l'applicazione client, quella server, il database server e gli altri soggetti della catena. Perciò è necessario applicare procedure di testing a vari livelli. Particolarmente delicata è la funzione degli script che gestiscono i dati, poiché rappresentano un'astrazione del database per le applicazioni client e server.

Due classi di problemi derivate dai bug nei database sono gli *integrity error* e gli *output error*. Gli integrity error possono essere visti come un danneggiamento o una perdita dei dati contenuti nei record; gli output error invece si hanno quando, nonostante i dati siano correttamente archiviati, non è possibile recuperarli. I sintomi evidenziati da entrambe le classi di bug sono simili, per questo è difficile capire di che tipo di errore si tratta (e dove è localizzato) con i test black-box. Vi sono invece numerosi tipi di test white-box che permettono l'individuazione di bug nei database con discreta efficienza. I test black-box rimangono comunque indicati per rilevare le anomalie, in quanto vengono eseguiti dal browser e dunque rispecchiano direttamente il comportamento del database dal punto di vista dell'utente.

3.3.4 Installation test

Questo tipo di test si concentra prevalentemente sull'eliminazione di tutta una serie di problemi che possono sorgere in fase di installazione dell'applicazione web:

- variabili di sistema non trovate implicano un'auto-configurazione scorretta;
- copie incomplete di file dovute a errori di trasmissione;
- incompatibilità hardware/software;
- interferenze da parte dell'utente in momenti critici;
- interferenze da parte di *firewall* ed antivirus.

La maggior parte delle applicazioni web prevede un'installazione solo dal lato server, ma ci sono anche casi in cui, viste le particolari funzionalità da svolgere, i comuni browser sono ritenuti non idonei e viene realizzato un apposito client. In queste eventualità i test devono riguardare anche il lato client dell'applicazione.

Testare i processi di installazione presuppone una solida conoscenza dei sistemi operativi e dell'ambiente in cui il tool di installazione opererà. Alcune delle principali operazioni svolte dal software di installazione (*installer*) sono:

- lanciare l'installazione dall'host sorgente;
- mantenere un log di installazione nell'host destinazione;
- recuperare le informazioni necessarie sull'ambiente destinazione ed installare l'applicazione secondo le opzioni scelte dall'utente;
- decomprimere eventuali file compressi;
- aggiornare il registro di sistema (Windows).

A queste operazioni è associato un insieme di possibili errori suddivisi in classi:

- *Functionality error*: quando l'installer fallisce il proprio obiettivo di completare l'installazione per i motivi più svariati (impossibilità di effettuare il *reboot*, mancanza di spazio su disco, impossibilità di creare directory e così via);
- *User interface design error*: quando l'interfaccia utente non rappresenta correttamente i risultati dell'installazione;
- *User interface implementation error*: quando l'interfaccia non implementa bene alcune funzionalità, quali ad esempio la semantica delle opzioni di setup propedeutiche all'installazione;
- *Misinterpreting collected information*: quando le informazioni raccolte sul sistema non sono quelle reali;
- *Operating system error*: quando avviene un errore del sistema operativo sottostante all'applicazione, durante l'installazione.

Terminata con successo l'installazione è comunque indicato verificare, con test black-box, che effettivamente tutto sia andato a buon fine.

D'altro canto, se si vuol offrire un servizio completo, è necessaria anche la presenza dei tool che si occupano della disinstallazione dell'applicazione (salvando eventualmente i dati dell'utente).

Un *uninstaller* dovrà provvedere a:

- rimuovere le directory create;
- rimuovere i file dell'applicazione;
- controllare se alcuni file usati dall'applicazione sono condivisi con altri software ed in caso affermativo informare l'utente;
- ripristinare lo stato dei registri di sistema (Windows).

Anche durante queste fasi possono sorgere numerosi errori. É naturale dunque che si debba procedere con test che verifichino l'effettiva eliminazione dell'applicazione.

Alcuni scenari di test che dovrebbero essere comunque considerati sono i seguenti:

- installare l'applicazione senza la configurazione minima richiesta;
- installare l'applicazione in un sistema operativo "pulito" oppure "affollato" da altre applicazioni;
- installare l'applicazione originale e successivamente i suoi aggiornamenti;
- controllare come l'installer reagisce se lo spazio su disco è insufficiente o l'installazione viene interrotta a metà.

3.3.5 Configuration e compatibility test

L'obiettivo dei test di configurazione e compatibilità è quello di trovare errori nell'applicazione mentre lavora negli ambienti più diffusi tra gli utenti. La strategia è spesso quella di eseguire un certo insieme selezionato di FAST, TOFT e forced-error test (benché sia difficile pilotare gli errori di ambiente nel Web), così da mettere alla prova la maggior parte delle caratteristiche del software nel contesto in cui si trova. Dalla parte del server si dovrebbero privilegiare i test che evidenziano le interazioni dell'applicazione con

- il web server stesso;
- il database server;
- il firewall;
- il sistema operativo;
- l'hardware;

- le applicazioni concorrentemente eseguite.

Dal lato client invece ci vuole particolare attenzione per

- i browser, di varie versioni;
- i sistemi operativi;
- i firewall "familiari'' e le applicazioni di limitazione dell'accesso dei bambini al Web;
- anti-virus;
- periferiche audio/video.

Infine, a prescindere dal lato dell'applicazione, è da verificare il corretto funzionamento con protocolli, periferiche e dispositivi di rete: TCP/IP, modem, schede di rete, bridge, router, hub e così via.

Sotto molti aspetti questo tipo di test è particolarmente costoso. É ovviamente impossibile considerare tutte le possibili combinazioni di hardware e software presenti sul mercato per i propri test di compatibilità! É necessaria invece un'apposita indagine statistica, che selezioni solo gli ambienti maggiormente diffusi tra gli utenti, sia del lato client che del lato server. É inoltre richiesta ai verificatori una buona conoscenza del comportamento e dei punti deboli delle maggiori tecnologie attualmente sul Web.

In genere, questo tipo di test, se deve essere eseguito su larga scala, viene affidato in *outsourcing* ad aziende specializzate. Esse dispongono di appositi laboratori per testare non solo le configurazioni più diffuse ma anche, a campione, quelle meno comuni.

3.3.6 Performance, load e stress test

Tra i problemi tipici dell'ambito di rete ci sono i ritardi e le interruzioni di comunicazione. Queste disfunzioni possono essere determinate da vari fattori: traffico nella rete, algoritmi inefficienti, errori di trasmissione e hardware non adeguato. É importante stabilire se un'applicazione web sa interagire con l'utente nei tempi previsti nella maggior parte dei casi e quali sono invece i limiti che non è consigliato oltrepassare.

I test di performance, load e stress sono legati tra loro e si distinguono più che altro per le diversa interpretazione dei risultati ottenuti: i test di performance mirano ad individuare una strategia per ottenere buone prestazioni dall'applicazione; i test di load puntano a valutare le performance con vari livelli di caricamento e utilizzo, per determinare oltre quale livello scadono

sostanzialmente; gli stress test infine hanno lo scopo di provare l'applicazione in condizioni di utilizzo estreme, quali l'utilizzo contemporaneo da parte di un elevato numero di utenti.

I test sono costituiti da simulazioni di un utilizzo "tipico" dell'applicazione da parte di centinaia o migliaia di utenti. Essendo impraticabile l'impiego di un tale numero di utenti umani diventa essenziale l'uso dei tool di testing automatico, i quali provvedono ad eseguire contemporaneamente le operazioni svolte da molti utenti virtuali.

Il risultato di questi test può essere positivo se le performance si mantengono accettabili rispetto ai livelli di utilizzo previsti, o negativo se invece l'applicazione (o l'ambiente nel quale si trova) è in difficoltà anche con un numero non esagerato di utilizzi concorrenti.

Con i test di stress è possibile individuare quale tra le seguenti risorse hardware tende ad esaurirsi prima:

- memoria;
- tempo di *CPU*;
- *banda passante*;

e dove invece si manifestano delle insufficienze software come:

- fallimenti causati da interrupt hardware;
- fallimenti legati alle operazioni in memoria;
- *deadlock*;
- problemi di multithreading.

É bene sottolineare l'importanza di questi test per tutte le applicazioni Web che provengono dalla "New Economy": per un sito come Amazon o per l'e-banking, più che per altri tipi di applicazione, le attese imposte all'utente pesano sull'immagine e sulla qualità complessiva del servizio. Un'applicazione che si prefigge di soddisfare il più largo bacino di utenza possibile (perché ogni utente è fonte di un introito economico), non può permettersi problemi di performance con bassi livelli di utilizzo.

Capitolo 4

Breve rassegna del software di web testing

Elencare a questo punto tutte le possibili alternative tra i tool di web testing *Open Source* e *proprietari* richiederebbe un tempo troppo lungo ed esulerebbe dagli scopi di questa tesi. É preferibile evidenziare solo alcuni software fra i più rappresentativi, che si distinguono per le soluzioni adottate o per le capacità di eseguire numerosi tipi di test. Si lascia al lettore il compito di approfondire le qualità dei singoli tool (peraltro in continua evoluzione), tramite la documentazione presente nei loro siti web. Come punto di partenza si segnala il seguente URL presso il quale è possibile trovare un elenco aggiornato delle principali utility per il web testing: **http://www.aptest.com/resources.html**.

- **JUnit**: JUnit è ormai uno standard nel software/web testing Java (ma non solo). Esso è costituito da un *framework* che consente di scrivere unit test ripetibili. Da JUnit derivano vari tool di testing che ne sfruttano la struttura, l'idea o le funzionalità: XMLTestSuite, JWebUnit, HTMLUnit, HTTPUnit ed altri ancora.

- **HTTPUnit**: HTTPUnit è una libreria Java disegnata ed implementata da Russell Gold. Consente di accedere ai siti web simulando l'uso di browser e per questo è molto impiegata anche in altri tool. Deriva a sua volta da numerosi pacchetti, tra cui JUnit, SAX, Tidy, XercesImpl e JS. Tra le funzionalità che mette a disposizione ci sono la compilazione dei form HTML, alcuni aspetti di autenticazione, navigazione negli

ipertesti, gestione degli oggetti HTML standard (tabelle, link, immagini, ...) e redirezione automatica. Non supporta alla perfezione il *parsing* di alcuni script, funzione che per altro può essere disabilitata. Permette invece di leggere le pagine web sia in formato testo che *XML DOM*. Bellerofonte, software presentato in questa tesi, si basa anch'esso su HTTPUnit per implementare i test di base (benché il verificatore sia libero di utilizzare pacchetti analoghi o più specifici per nuovi test).

- **Solex**: Solex è un tool per i test funzionali fornito come plug-in della piattaforma di sviluppo Eclipse. Solex fa parte dei tool di testing "record/replay": ha la particolarità di poter registrare le sessioni client agendo come un server proxy, interponendosi cioè tra il browser ed il server, in ascolto delle richieste e delle risposte HTTP. In seguito le sessioni registrate possono essere rieseguite, con la possibilità di assegnare dei nuovi parametri alle variabili contenute nei messaggi HTTP.

- **PureTest**: PureTest, unita a PureLoad, fornisce una suite di testing piuttosto completa, comprendente un editor grafico di scenari e un *WebCrawler* per l'analisi statica della struttura di un sito web. Oltre che con le funzionalità disponibili su HTTP, PureTest è compatibile con la maggior parte dei protocolli standard in uso, tra cui NNTP, FTP, SMTP, IMAP, JDBC, LDAP, DNS e JMS. É permessa la creazione di scenari sia con il metodo "record/replay" sia con quello "data driven" ovvero editandoli in funzione dei contenuti da testare. PureTest consente l'automazione integrata con Ant di Jakarta ed offre anche un'interfaccia dalla riga di comando per richiamare il tool da altri script.

- **Siege**: Siege è un tool Unix per eseguire stress test e "mettere sotto assedio'' un web server. Supporta autenticazione, cookies ed i protocolli HTTP e HTTPS.

- **TestMaker**: è un tool open source Java rivolto particolarmente al testing delle performance dei Web Service. Dispone di un ambiente grafico di scrittura/esecuzione di test e di un apposito linguaggio di scripting, ispirato a Python, che consente di comunicare agevolmente con SOAP, HTTPS, .NET, JSP. Ha infine la possibilità di eseguire i

test creati in modo concorrente, per simulare un utilizzo condiviso del Web Service.

- **JSpider**: è una flessibile implementazione Java di un *web spider*, ovvero un agente che, muovendosi nel Web (proprio come un ragno sulla ragnatela), raccoglie determinate informazioni sulle pagine incontrate. JSpider consente quindi di verificare automaticamente lo stato di link ed altri elementi web, di controllare se ogni risorsa è raggiungibile, effettuare copie locali di interi siti e, in generale, effettuare test di caricamento e performance.

- **Canoo WebTest**: è un tool per i test funzionali basato su HTTPUnit che permette di editare i propri scenari di test direttamente in XML. Esso si integra in modo interessante con ANT, tool di automazione del progetto Jakarta, per creare test suite facilmente combinabili tra loro.

- **HTML Validation Service**: è un servizio gratuito di validazione online delle pagine HTML e XHTML, offerto dal consorzio W3C per le standardizzazioni nell'ambito Web.

Parte II

Web testing con Bellerofonte

Capitolo 5

Presentazione del software

In questo capitolo si descriverà l'architettura di Bellerofonte, software di web testing realizzato nell'ambito di questa tesi. Si intende così fornire ad eventuali altri sviluppatori le informazioni necessarie per poterlo estendere secondo le proprie esigenze. Nel prossimo capitolo invece è stata inserita una descrizione più pratica, comprendente un manualetto d'uso, per consentire al semplice utente di apprendere rapidamente l'utilizzo del programma.

5.1 Descrizione ad alto livello

5.1.1 Dal mito al software

Prima di descrivere ciò che Bellerofonte si propone di raggiungere e come cerca di farlo, si spenderà qualche parola sul suo nome e sul mito al quale si ispira.

La cultura greca ci ha tramandato un numero considerevole di eventi mitici con forti contenuti simbolici. Il mito di Bellerofonte, eroe greco, racconta, tra le altre sue vicissitudini, della lotta con la *Chimera*, il mostro con tre teste: leone, capra e serpente. L'eroe riuscì nell'impresa di sconfiggere la Chimera grazie alla sua abilità ed a Pegaso, il cavallo alato, grazie al quale poté colpirla dall'alto.

La Chimera nella nostra cultura continua a simboleggiare ciò che è insormontabile, difficile da raggiungere o da sconfiggere.

Nell'ambito del web testing la Chimera è rappresentata dai bug e dalla speranza di scovarli tutti. Un software che dunque si proponga di "lottare'' con i bug, non può non identificarsi in Bellerofonte. Per concludere l'analogia,

Pegaso sarà l'interfaccia di Bellerofonte, cioè lo strumento che ne facilita la lotta.

5.1.2 Cosa è Bellerofonte

Si deve subito premettere che Bellerofonte è un software pensato per essere usato da sviluppatori e verificatori. Non possiede di per sé particolari accorgimenti che ne consentano un facile uso da parte di utenti generici, ma punta ad essere semplice per gli utenti specializzati. Per innalzare il livello di usabilità dovrebbe essere sviluppata un'interfaccia indipendente (Pegaso appunto), grafica o meno, che guidi l'utente nell'editing dei file di cui Bellerofonte ha bisogno.

Come si è mostrato sinteticamente nel capitolo quattro, esistono già molti tool e librerie a supporto del web testing. Ci sono però delle critiche che si possono sollevare a molti software in commercio:

- alcune librerie consentono di creare solo dei test case "statici", cioè relativi alla particolare caratteristica da verificare e difficilmente adattabili a nuovi contesti;
- spesso, dove è permessa l'estrazione e la contestualizzazione delle variabili del test, non è però consentita l'integrazione con altre librerie;
- pochi tool dispongono di un'interfaccia valida dalla riga di comando e dunque sono difficilmente richiamabili da uno script esterno;
- nonostante il Web accomuni molti tipi di sistemi operativi diversi, alcuni tool sono progettati per essere eseguiti solo su un gruppo ristretto di essi. Analogamente ci sono dei software che possono testare solo le applicazioni web inserite in determinati ambienti, il tutto a scapito della portabilità;
- scrivere un nuovo tipo di test, relativo ad una particolare funzionalità da testare, non è sempre permesso o può essere difficile integrarlo ai test già esistenti;
- la descrizione dei test case da far eseguire al tool può non essere immediata e sintetica o può presupporre conoscenze avanzate.

Presa coscienza dell'esistenza di questi punti deboli nei software di web testing, si è cercato, per quanto possibile, di eliminarli in Bellerofonte, che si propone dunque come uno strumento di analisi *data driven* per la rilevazione di bug in siti web. Esso offre un insieme di test "atomici", componibili fra

loro, capaci di emulare la navigazione in un sito da parte di un utente che faccia uso di un browser web.

Bellerofonte si inserisce tra gli strumenti di web testing automatici, utilizzabili nella fase di analisi dinamica della verifica. Con questo software si vuol rendere agevole sia la programmazione di specifici test da parte degli sviluppatori (parallelamente alla realizzazione della nuova funzionalità), sia l'esame ripetuto nel tempo da parte dei verificatori. É quindi un software adatto anche a forme di web testing regressivo.

I test eseguibili sono essenzialmente di tipo gray-box, funzionale, che non guarda tanto alla sintassi di ciò che esamina quanto alla sua rappresentazione tramite il client. Si può parlare di testing gray-box in quanto è necessaria, da parte dei realizzatori di un test, anche una conoscenza diretta (di tipo white-box) dell'elemento da testare: saper identificare nomi, gerarchie e relazioni interne è essenziale per produrre test mirati ed efficaci.

Dato che la tecnologia di rete è in continua evoluzione e le funzionalità disponibili sul Web sono le più disparate, uno dei principali obiettivi di Bellerofonte è quello di raggiungere un certo grado di flessibilità nella tipologia dei test eseguibili. La sua architettura è dunque stata pensata per permettere una facile integrazione di test atomici scritti ex-novo, utilizzando magari librerie innovative, diverse da quelle usate nei test precedentemente implementati.

I test atomici, che in seguito saranno definiti "Single test" o test singoli, possono d'altra parte essere di complessità molto variabile: da una banale verifica del titolo di una pagina web alla più complicata visita in profondità di un sito. Rispettando alcune caratteristiche di base è quindi possibile aggiungere a Bellerofonte test specifici compresi in categorie di web testing diverse da quella strettamente funzionale.

5.1.3 Gerarchia dei test eseguibili

Bellerofonte gestisce i test a due livelli diversi:

- i *Single test* sono i test atomici componibili citati in precedenza, i quali corrispondono semanticamente ai test case definiti nel capitolo due. Essi sono costruiti sulla base delle funzionalità messe a disposizione dai pacchetti Java di web testing (per esempio HTTPUnit), da quelle presenti in altri tool eseguibili dalla riga di comando o da una loro combinazione. Un test singolo si incarica della verifica di un'unica caratteristica dell'applicazione web. Esempi di Single test sono: l'individuazione di un determinato link, la navigazione attraverso esso, il controllo di un titolo di pagina, la scrittura in un campo di un form e

così via. I Single test non sono direttamente eseguibili: devono essere inclusi in un Macro test (si veda il punto seguente) per contestualizzarne e gestirne gli input e gli output.

- i *Macro test* non sono altro che insiemi strutturati di Single test e per questo il loro ruolo si avvicina a quello dei test script. Essi rappresentano una sequenza logicamente coerente di Single test, volta a simulare l'interazione di un browser con un sito web. Ogni Single test si preoccupa di effetture solo dei controlli limitati, mentre invece lo svolgimento di un Macro test, inteso come insieme di test singoli, si traduce in un'analisi complessiva del sito così come lo "vedrebbe" un utente. Il Macro test non è solo un contenitore di Single test, ma, come si vedrà, ha anche svariati compiti di coordinazione.

Il compito di un verificatore è quello di definire Macro test a partire da Single test preesistenti o appositamente implementati.

5.2 Dettagli implementativi

Evitando di procedere ad un esame diretto del codice, per altro commentato, si illustreranno adesso i dettagli strutturali e le scelte implementative fatte per realizzare Bellerofonte.

5.2.1 Linguaggio e strumenti di sviluppo

Bellerofonte è stato realizzato in Java. La scelta è caduta su questo linguaggio perché è particolarmente indicato per la realizzazione di applicazioni di rete ed ha un buon grado di portabilità. Nella fattispecie sono disponibili per Java numerosi pacchetti (o collezioni di classi) rivolti al testing di vario genere e dunque anche al web testing. I pacchetti più usati, tra quelli destinati al testing di applicazioni web, sono HTTPUnit, JWebUnit e XMLTestSuite, i quali offrono astrazioni molto comode per operazioni di rete di basso livello.

Il meccanismo di compilazione ed interpretazione run-time di Java risulta penalizzante se si guarda alle prestazioni che un software deve garantire in certi contesti. Nel caso di Bellerofonte però le prestazioni hanno un'importanza relativa, in quanto i tempi di esecuzione dipendono prevalentemente dallo stato della rete, piuttosto che dall'efficienza del software. É invece essenziale assicurare la portabilità di un software, quando non si conosce su quale tipo di piattaforma sarà impiegato (Windows, Unix, Mac-OS e così via). Bellerofonte nasce e sarà utilizzato in ambito universitario e quindi

deve garantire quanta più portabilità possibile. Java, tramite l'interpretazione del *byte-code* generico da parte delle *Java Virtual Machine* specifiche, consente di svincolarsi dalla specifica piattaforma di esecuzione.

Per scrivere il codice di Bellerofonte si è utilizzato l'editor per Windows JCreator ver. 2.5, che si integra con l'ambiente di sviluppo della Sun Java 2 SDK 1.4.0.03. Il funzionamento è quindi garantito solo con versioni di interpreti Java uguali o superiori all'1.4.

5.2.2 Utilizzo di pacchetti esterni

Per implementare i Single test da gestire ed eseguire con Bellerofonte, sono stati usati pacchetti Java aggiuntivi (open source). A loro volta i pacchetti importati dipendono spesso da altri pacchetti che in questa sede non verranno descritti.

- **com.meterware.httpunit**: Bellerofonte è stato basato su questo pacchetto in quanto, tra quelli relativi al web testing in Java, è il più utilizzato e completo. Oltre che per la realizzazione di gran parte dei Single test, ci si riferisce ad HTTPUnit per la gestione delle opzioni ed il passaggio di alcuni parametri tra un Single test e l'altro. Benché Bellerofonte non possa prescindere da HTTPUnit, è stato fatto il possibile per limitarne la dipendenza a livello di Single test. Legarsi ad un unico pacchetto è infatti un approccio troppo restrittivo ed inoltre, mostrando HTTPUnit lacune sotto certi aspetti (test di scripting), si vuol consentire l'uso di alternative. L'URL di riferimento per questo pacchetto è http://www.httpunit.org/ oppure presso sourceforge.net;

- **com.stevesoft.xmlser**: Xmlser è un pacchetto (sempre Open Source) che offre API in grado di *serializzare* e *deserializzare* automaticamente oggetti in XML. XML è un formato di codifica universale dei dati, attualmente in grande diffusione a tutti i livelli applicativi. Nell'ambito di Bellerofonte è impiegato per serializzare gli oggetti che contengono Macro test, in modo da renderli comunque facilmente editabili ed eventualmente importabili. L'url di riferimento è http://www.javaregex.com/;

- **java.util.regex**: Regex è un pacchetto contenuto in Java 2 Standard Edition (versione 1.4) di Sun. Esso consente e facilita la gestione di espressioni regolari per la ricerca di pattern in un certo testo. In Bellerofonte è stato impiegato all'interno di alcuni Single test per consentire la selezione di un determinato insieme di oggetti in base al loro identificatore. Un suo utilizzo tipico è nella selezione dei link da seguire.

Per maggiori informazioni si consiglia di effettuare una ricerca presso il sito http://java.sun.com/;

- **org.htmlparser**: HTMLParser viene presentato come un'evoluzione di HTTPUnit (dalla quale discende), in quanto offre le stesse funzionalità ma in modo semplificato e maggiormente organizzato. Ad esempio, all'interno di HTMLParser, sono presenti dei "robot" configurabili dall'utente per eseguire determinate operazioni automatiche. Una delle sue qualità migliori (che giustifica peraltro il suo nome) è la facilità con cui consente di estrarre informazioni dal codice HTML. Nel caso di Bellerofonte si è usato per analizzare del semplice testo. Il sito di riferimento è http://htmlparser.sourceforge.net .

5.2.3 Identificazione delle classi

Come mostra il diagramma UML di figura 5.1 le principali classi del programma sono nove (il carattere "_'', pur presente nei nomi di alcune classi, è stato omesso in alcune descrizioni per semplicità):

1. **Web_test**: questa classe è, dal punto di vista dell'utente, la più importante. In essa risiede il metodo "main'' che si preoccupa di catturare i parametri in ingresso. In Web test si avvia e si coordina tutto il processo di creazione ed esecuzione dei Macro test. La classe Web test possiede un oggetto della classe Hdd interface, che funge da interfaccia per il caricamento e il salvataggio dei dati sull'hard disk;

2. **Macro_test**: la classe Macro test è invece quella che identifica e contiene tutte le componenti di un test. Infatti essa ha un vettore di oggetti di classe Single test, un altro vettore di oggetti Report (che in realtà è gestito in modo da contenerne al massimo due), un oggetto di classe TestOption ed un oggetto di classe Report contenente il risultato dell'ultima esecuzione del Macro test. Alla classe Macro test spetta dunque la responsabilità di istanziare i test singoli su richiesta ed eseguirli, leggere le opzioni e rappresentare i risultati delle esecuzioni;

3. **Single_test**: La classe Single test è l'unica *classe astratta* del programma e serve da modello per tutti i test singoli concreti che devono necessariamente ereditare da lei. La classe Single test possiede infatti i campi ed i metodi comuni ad ogni test singolo, il quale deve specializzarli concretamente affinché possano essere eseguiti. Un Macro test contiene un numero maggiore o uguale a zero di oggetti Single test

e nell'eseguirli non distingue tra l'uno e l'altro, eccezion fatta per il particolare test "FollowLinks";

4. **Hdd_interface**: si è già accennato alla funzione di questa classe. In particolare essa conosce l'indirizzo sull'hard disk dei vari file necessari a Bellerofonte ed è pertanto deputata al reperimento ed al salvataggio delle informazioni. Essa rappresenta per la classe Web test un'astrazione rispetto a tipiche problematiche di codifica e decodifica, di cui si fa carico;

5. **Report**: la classe Report è usata in ogni test singolo per "annotare'' i risultati ottenuti. Ogni report deve essere strutturato e coerente con i report di altre esecuzioni, così da permettere eventuali confronti. Si delega quindi ai report la funzione di formattare e conservare in modo adeguato i risultati di ogni Single test. Un report rimane comunque un oggetto di Macro test, ma viene passato tra i vari test singoli come il "testimone'' di una staffetta;

6. **TestOption**: le opzioni rappresentano per il Macro test la definizione dell'ambiente nel quale si dovrà svolgere. Un oggetto di classe TestOption appartiene sempre ad un Macro test e non può essere modificato direttamente dai test singoli. Le opzioni contengono l'URL iniziale, l'abilitazione o meno di cookie e redirect ed in generale ogni opzione configurabile tramite HTTPUnit. I parametri selezionati nelle opzioni influiscono sulle modalità di lettura e rappresentazione della pagina web in Web page;

7. **Timer**: questa classe era preesistente a Bellerofonte ed è stata importata nel progetto da un corso di laboratorio dell'Università di Berkeley. Essa fornisce ovviamente le funzionalità tipiche di un timer (start, stop, elapsed) per misurare la durata dei Single test e del Macro test nel complesso. Utilizza il timer di sistema ed approssima male intervalli di tempo troppo brevi;

8. **Web_page**: la classe Web page è stata introdotta per avere, all'interno di Macro test, un unico e coerente riferimento alla pagina web analizzata in un dato momento. Web page contiene un vettore di oggetti in grado di descrivere, raggiungere e manipolare la relativa pagina. Questi oggetti sono normalmente soltanto quelli appartenenti al pacchetto HTTPUnit (WebResponse, WebConversation e ClientProperties) ed occupano nel vettore posizioni predefinite, note ad ogni test singolo. Un test singolo che volesse far uso di oggetti diversi per rappresentare

la pagina web, dovrebbe poi memorizzarli sul vettore in modo da non sovrascrivere le locazioni già occupate. Al fine di rendere visibili gli effetti di un test anche a quelli che lo seguono (si pensi ad esempio al riempimento di un modulo), Web page rientra tra gli oggetti che Macro test trasferisce tra un test singolo e l'altro;

9. **Macro test_recursive**: questa classe è invocata da Macro test dopo l'esecuzione del test singolo "FollowLinks", in quanto si rende necessario ripetere ogni test singolo rimanente in ogni link seguito. In altre parole implementa un *fork* nel cammino, altrimenti sequenziale, dei test singoli eseguiti.

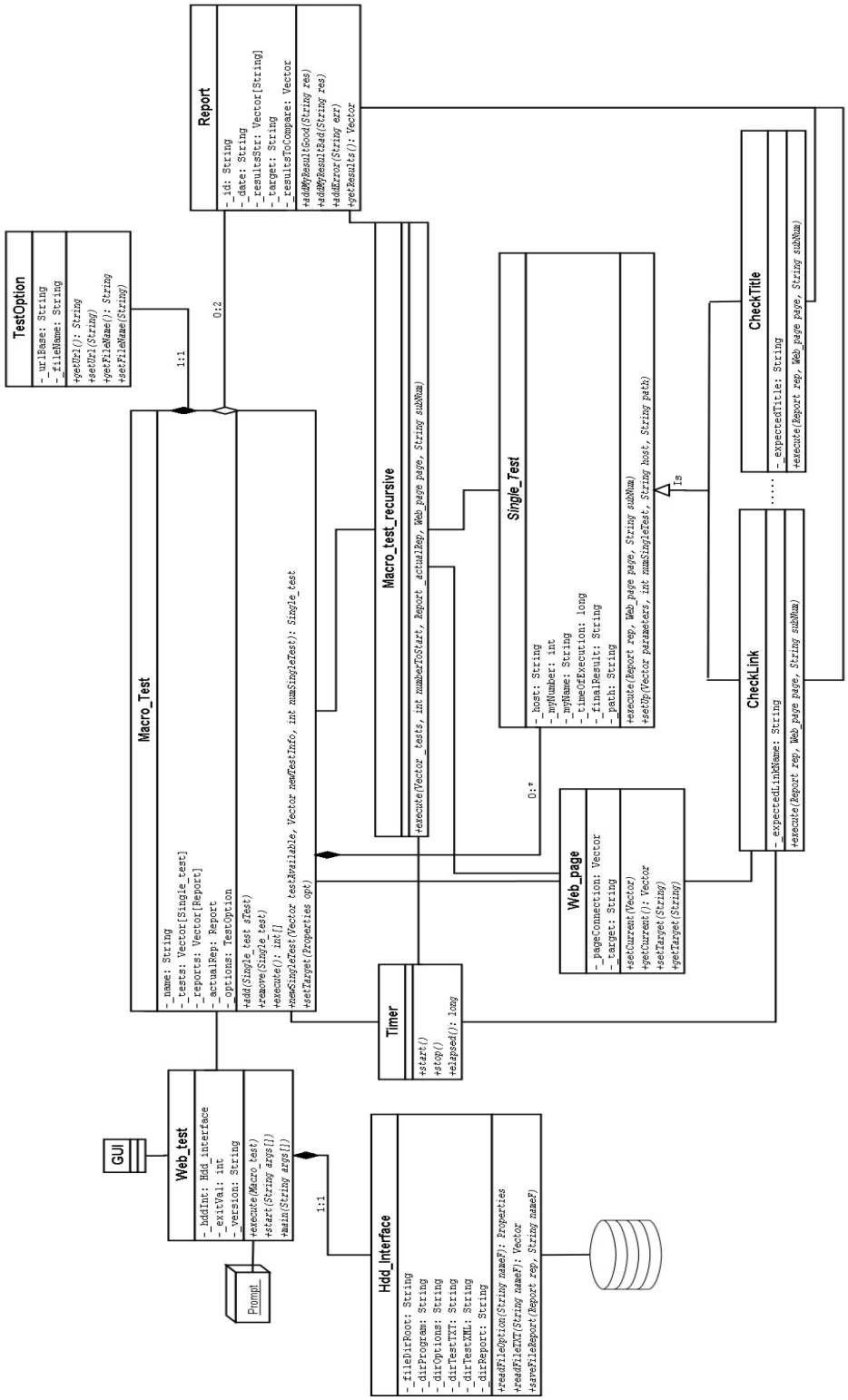

Figura 5.1: Diagramma UML di Bellerofonte

5.2.4 Analisi dei metodi più significativi

Entrando un pò più nel dettaglio si descrivono ora tre importanti metodi appartenenti alle classi presentate nel paragrafo precedente.

- *Web test.start*: questo è sicuramente il metodo più generale. Viene invocato dal main sull'oggetto di classe Web test e coordina tutte le azioni principali svolte durante il test. In input prende il vettore di argomenti passati al programma dalla riga di comando. Se manca un argomento necessario al funzionamento, start lancia una console interattiva che richiede all'utente tutti i parametri necessari oppure fa terminare il programma. In base a questi parametri (nome del file delle opzioni, nome del file test e modalità di salvataggio del report) start agisce in modi differenti. Provvede anzitutto a richiedere all'Hdd interface il contenuto del file contenente i test singoli attualmente riconosciuti (ovvero le cui classi sono presenti ed istanziabili). Poi crea un oggetto di classe Macro test e gli assegna il nome del file test ricevuto come parametro. Prosegue col richiedere l'elenco delle opzioni all'oggetto Hdd interface, a cui passa il relativo nome di file. Se il file test letto dall'hard disk ha estensione ".tst'' significa che è codificato in XML e rappresenta un Macro test precedentemente eseguito e serializzato. Si può dunque effettuare una deserializzazione ed un ripristino rapido del suo stato, provvedendo solamente alla sovrascrittura delle opzioni (dato che potrebbero essere cambiate dall'ultima esecuzione). Altrimenti, se il file test ha un'altra estensione (".txt''), start capisce che si tratta di un file di testo che rappresenta un Macro test alla prima esecuzione. Provvede dunque a tradurre l'elenco di direttive ricevute in un oggetto Macro test pronto per essere eseguito: imposta le opzioni, legge i nomi dei test singoli che dovranno comporre il Macro test, li crea e glieli aggiunge. Terminata questa fase preparatoria Macro test viene finalmente eseguito.

  Una volta che ogni Single test è terminato (con vario esito) ed ha aggiunto al report il proprio risultato, start cerca nel terzo parametro (opzionale, passatogli dalla riga di comando o dalla console interattiva), le indicazioni su come trattare l'oggetto Macro test ed il relativo report. Prima di terminare start visualizza il conteggio dei test singoli eseguiti, suddividendoli in base all'esito riportato. Bellerofonte si basa poi su questi conteggi per decidere il proprio valore di ritorno al sistema: 0 nel caso in cui ogni Single test sia stato passato, 1 se si è verificato almeno un fallimento e 2 se si è verificato almeno un errore generale;

- *Macro test.newSingleTest*: questo metodo di Macro test istanzia, a tempo di esecuzione, nuovi oggetti corrispondenti ai test singoli (per esempio CheckTitle), continuando a gestirli tramite la loro classe base: Single test. Viene sfruttato cioè il *polimorfismo*, una delle caratteristiche principali dei linguaggi orientati agli oggetti. In questo caso ogni test singolo concreto può essere istanziato ed utilizzato al posto della classe base astratta Single test, da cui deriva. Nel far ciò i metodi della classe derivata (concreta) si specializzano, sovrascrivendo i metodi della classe base (setUp ed execute principalmente). Macro test, per poter creare test singoli concreti, deve conoscere i nomi delle rispettive classi. Una volta ottenuti i nomi (leggendo da un apposito file di programma), Macro test crea altrettanti oggetti grazie alla *Reflection* di Java. Se per ogni nome di test è stato possibile creare il Single test concreto richiesto, questo è restituito all'ambiente chiamante correttamente inizializzato e pronto per essere eseguito;

- *Macro test.execute*: l'execute di un Macro test corrisponde all'esecuzione di ogni test singolo presente nel vettore _tests più varie operazioni di coordinamento. Anzitutto viene creato un nuovo Report, che sarà aggiornato in ogni fase dell'elaborazione. In seguito si fa partire il Timer e si cominciano ad eseguire sequenzialmente i test singoli. Ad ogni test singolo viene passato:

  – il riferimento al Report da aggiornare;
  – un oggetto Web page al quale riferirsi per i test;
  – il numero di sequenza del Single test.

  Una volta terminata questa fase viene fermato il Timer (rilevando il tempo di esecuzione) e viene ritornato all'ambiente chiamante il numero di test passati, falliti e gli eventuali errori generati.

5.2.5 Rassegna dei test base implementati

Nella versione base di Bellerofonte sono stati realizzati 22 test singoli. Eccoli descritti brevemente (per ulteriori informazioni ci si può avvalere del capitolo sei e dei documenti inclusi nel pacchetto orbilio.webTest.bellerofonte):

1. **AddCookie**: con questo test è possibile definire il valore di un cookie da inviare al server per identificare una sessione;

2. **CheckButtonValue**: controlla se un form ha un certo pulsante (per esempio "Submit'') ed eventualmente lo "clicca'';

3. **CheckCharacterSet**: controlla il set di caratteri usato in un sito;
4. **CheckForm**: controlla se in una pagina è presente un certo form;
5. **CheckFormAction**: controlla il valore del parametro "action'' di un form;
6. **CheckFormContent**: verifica se un form ha un campo con un dato valore;
7. **CheckFormField**: verifica la presenza di un campo in un form;
8. **CheckFormMethod**: controlla il valore del parametro "method'' di un form;
9. **CheckImage**: verifica la presenza di un'immagine;
10. **CheckLink**: controlla la presenza di un link e la validità dell'URL associato;
11. **CheckMailLinks**: esamina un link "mailto'';
12. **CheckTable**: verifica la presenza di una data tabella;
13. **CheckTableContent**: data una tabella, verifica il contenuto di una sua cella;
14. **CheckTextInPage**: data una parola chiave, la ricerca nel testo contenuto in una pagina;
15. **CheckTitle**: verifica il titolo della pagina;
16. **FollowLinks**: data una stringa od un URL seleziona tutti i link che gli corrispondono e li restituisce a Macro test affinché li possa seguire;
17. **GetFrameContent**: se nella pagina è presente un frame con un dato nome, passa ad esaminarlo;
18. **GetHttpHeader**: riporta i contenuti attuali delle intestazioni nei messaggi HTTP;
19. **ReachAll**: selezionata una profondità di visita ed un elenco di pagine da raggiungere, verifica la loro raggiungibilità effettuando una visita in profondità del sito. Non segue i link esterni;
20. **SetFormContent**: imposta un certo valore in un campo di un form;

21. **SubmitForm**: preme il bottone "Submit'' di un form e lo invia al server, aggiornando Web page con la risposta ricevuta;

22. **UploadFileWithForm**: dato un form in cui è prevista questa funzione, effettua un upload di un file.

5.3 Estendere Bellerofonte

Concepire un software statico, difficilmente estendibile, in un settore tanto mutevole quale è il web testing, è un grave errore di valutazione: troppo arduo pensare di racchiudere in un'unica soluzione ogni possibile test effettuabile, tanto più che non si sa cosa dovrà essere testato in futuro. Un software difficilmente estendibile, benché completo e curato sul momento, rischia di avere vita breve, perché scarsamente adattabile alle particolari esigenze dei verificatori. Con Bellerofonte non si ha la presunzione di offrire un prodotto "finito'', bensì un "progetto in continua evoluzione'', capace di integrare con poco sforzo eventuali nuovi test.

5.3.1 Come aggiungere funzionalità ex-novo

Aggiungere semplicemente dei test a quelli già esistenti, senza portare modifiche strutturali o semantiche alle classi di Bellerofonte, dovrebbe risultare un compito relativamente semplice per degli sviluppatori. Disponendo della directory di lavoro di Bellerofonte (figura 6.1), i passi da eseguire sono standard:

1. anzitutto si deve individuare la libreria adatta alla realizzazione del nuovo test: o si sfrutta solo la libreria di default (HTTPUnit) o se ne deve importare (in aggiunta) una nuova. Se la libreria scelta non è compresa da quelle già utilizzate in precedenti test allora è necessario:

   (a) copiare il relativo file jar nella directory /jars;

   (b) editare il file manifest.mf in /lib aggiungendo nella riga "Class-Path'' la scritta "../jars/'' + <nome Jar aggiunto>

   (c) ricordarsi di aggiungere il nuovo archivio jar nel Path dell'editor usato per scrivere il nuovo test;

2. una volta selezionata una o più librerie adatte, aprire il file "Anonymus.java'' che si trova in /src e salvare subito con altro nome il nuovo test. Anonymus è semplicemente lo "scheletro" di ogni nuovo test singolo e dovrebbe pertanto rimanere tale;

3. rinominare la nuova classe ed il suo costruttore con il nome del nuovo test. É importante che ogni classe test resti estensione di Single test ed implementi l'interfaccia "Serializable'';

4. viene adesso la fase più complicata. Nella nuova classe test sono individuabili cinque aree personalizzabili. Si deve procedere al loro corretto riempimento evitando di modificare il restante codice, che serve a Bellerofonte per gestire i test singoli a prescindere dalla loro funzione. In un certo senso il programmatore dovrà comportarsi come se stesse sovrascrivendo determinate parti di un'*interfaccia*. Il significato di ogni area è spiegato più dettagliatamente nei documenti "howTo" inclusi nella directory /various. Qualora la funzione di un'area non risultasse chiara è consigliabile l'ispezione del listato di qualche altro Single test già implementato.
   Il listato della classe *Anonymus*, così come viene trovata dal programmatore nel momento in cui decide di specializzarla in un nuovo test singolo, è presentato di seguito. Le aree sulle quali si dovrà intervenire sono delimitate da apposite stringhe di commento:

Listato della classe Anonymus.java

```
package orbilio.webTest.bellerofonte;

import java.util.Vector;
import java.io.Serializable;

//////////////// FOLLOWING CAN CHANGE FROM TEST TO TEST
import java.net.*;
import java.io.IOException;

// Libraries used
import com.meterware.httpunit.*;
//////////////// END OF COMMON CHANGE AREA

/**
 * @author Claudio Tortorelli - 2003
 *
 */
//-----------------------------------------

class Anonymus extends Single_test implements Serializable
{
/**
 * Fields
 */
```

```
//////////////// FOLLOWING CAN CHANGE FROM TEST TO TEST
  private String _expectedInput = "";
//////////////// END OF COMMON CHANGE AREA

/**********************************************
 * Constructor
 */
  public Anonymus()
  {
   super();
  }
/**********************************************
 * This setUp method reads an ordered list of
 * parameters embedded in a Parameters object, some
 * test options and the single test's number. With
 * these informations it intializes its objects.
 *
 * IN: a vector with parameters specified in
 * macro test file and the number of test.
 * The vector has at first position the test's
 * name and the other positions hold the actual parameters
 * OUT: void
 */
  public void setUp(Vector parameters, int numSingleTest)
  {
   _myNumber =  numSingleTest;
   _myName = (String)parameters.elementAt(0);
   _finalResult = "notCheckedYet";
   _timeOfExecution = "0";

//////////////// FOLLOWING CAN CHANGE FROM TEST TO TEST
  //read words from file test txt or xml
   _expectedInput = (String)parameters.elementAt(1);
//////////////// END OF COMMON CHANGE AREA

  }
/**********************************************
 * This method implements the particular execution of
 * this single test. It take as parameters the
 * report and the list of values returned by some
 * eventual previous tests. It returns its values.
 *
 * IN: report for add results, file name target,
 * vector with results of previous test
 * OUT: vector with the results of this test
 */
```

```
  public void execute(Report rep, Web_page page, String subNum)
  {
  // strings of header of Single_test
    String num = _myNumber + subNum;
    String target = page.getTarget();
    String head1 = "TEST: \""+_myName+"\""+" | Url: "+target;
    rep.addHeader(head1, num);
  // start timer
    Timer tim = new Timer();
    tim.start();

//////////////// FOLLOWING CAN CHANGE FROM TEST TO TEST
    rep.addResultExpected("Input: " + _expectedInput);
//////////////// END OF COMMON CHANGE AREA

  // get page's state elements,
  // if 0 then the page was not loaded correctly
    Vector pageData = page.getCurrent();
    if (pageData.size() != 0)
    {

//////////////// FOLLOWING CAN CHANGE FROM TEST TO TEST
/* //test passed to Report
 *  rep.addMyResultGood("The file upload has been setted");
 * // don't change: it is used for comparison
 *  _finalResult = "PASSED";
 *
 * //test failed to Report
 *  rep.addMyResultBad("The file upload has been setted");
 * // don't change: it is used for comparison
 *  _finalResult = "FAILED";
 */
//////////////// END OF COMMON CHANGE AREA

    }
  // the web page wasn't load correctly
    else
    {
      rep.addError("Impossible to perform this test");
      _finalResult = "ERROR";
    }
  // stop timer and set elapsed time
    tim.stop();
    long oneSec = 1000;
    _timeOfExecution = tim.elapsed()/oneSec+"."+tim.elapsed();
    rep.addSecOfExecution(_timeOfExecution);
  // add to Report the significative datas
```

```
    rep.addDataToCompare(_myName, _finalResult, _timeOfExecution);
  // end of execution
    rep.addSeparation();
  }
} // end of class
```

5. quando il test realizzato ha complessità elevata si consiglia di realizzare dei metodi privati interni, da richiamare nelle aree sopra descritte. Come esempio si può considerare il metodo *visitDep* del test *ReachAll*.

   Seguendo lo stesso principio, se per realizzare il test si devono implementare svariate classi accessorie, è indicato includerle in un'apposita libreria da inserire assieme agli altri jar nella directory /jars. Anche in questo caso può essere preso come esempio ReachAll, per il quale è stata realizzata la libreria claudiosoft.struct.btree;

6. se, al termine dell'attività di programmazione, il codice Java del nuovo test compila correttamente, si deve procedere all'aggiunta del file al progetto orbilio.bellerofonte. Una volta effettuato un *rebuild* dell'intero progetto, si deve verificare che l'output della compilazione (ovvero i file ".class'' tra cui quello del nuovo test) sia stato correttamente inserito in /orbilio/webTest/bellerofonte. É in quella directory che Bellerofonte andrà a cercare la definizione del nuovo test invocabile;

7. si procede poi ad editare il file "TestList.txt'' contenuto in /webtest-files/program. Esso contiene un elenco dei nomi di tutti i Single test per i quali esiste una definizione. Per aggiornarlo basta aggiungere in fondo alla lista dei test attualmente riconosciuti, il nome del nuovo test creato. É importante che sia rispettata l'esatta sequenza di maiuscole e minuscole;

8. editare anche il file "TestSynopsis.txt'' in /webTestFiles/program (e il suo omologo ".doc'' in /various) aggiungendo l'esatta segnatura ed una sintetica descrizione del nuovo test. Ciò non è strettamente necessario

al funzionamento del software, ma è richiesto per lasciare una minima documentazione sui nuovi test introdotti.

9. eseguire infine lo script batch "makeJar.bat'' in /bin, il quale provvede automaticamente ad:

   - aggiornare il file "Bellerofonte.jar'' in /lib aggiungendo il nuovo file ".class'';
   - aggiungere alla variabile di sistema Classpath il nome di eventuali nuovi archivi jar inseriti in /jars.

5.3.2 Possibili sviluppi futuri

Attualmente Bellerofonte non dispone di un'adeguata interfaccia grafica (Pegaso) che dovrebbe supportare l'utente durante tutta le fasi di preparazione ed esecuzione del test descritte nel prossimo capitolo.

Un'altra mancanza che si potrebbe pensare di colmare è l'impossibilità di effettuare test di performance e stress. Essendo però Bellerofonte richiamabile dalla riga di comando, è piuttosto semplice creare uno script esterno che includa una sua esecuzione separata in *n thread* diversi. D'altro canto, Bellerofonte è stato progettato principalmente in funzione di Orbilio, applicazione che attualmente non ha bisogno di simili test.

Altri aspetti che meriterebbero maggiore attenzione in versioni future di Bellerofonte potrebbero essere:

- ottimizzazione del disegno e del codice;
- migliore gestione dei report;
- migliore gestione del timer;
- utilizzo più raffinato dell'XML nella definizione dei Macro test;
- inserimento di test più sicuri nell'analisi degli script;
- considerare specifici test di usabilità, quali quelli che verificano l'accessibilità per utenti disabili.

Capitolo 6

Manuale d'uso

In questo capitolo verranno illustrate le procedure di utilizzo di Bellerofonte. Prima però è bene chiarire quali sono e come sono strutturati i file e le directory usate dal programma.

6.1 File di programma

Bellerofonte presume di lavorare sempre all'interno di un sistema predefinito di file e directory rappresentato in figura 6.1.

Il programma fa sempre riferimento alla directory webTestFile come propria *root*, la quale a sua volta ha cinque sottodirectory:

- *options*: contiene i file che specificano le impostazioni per l'esecuzione dei file test;
- *program*: vi sono i file necessari al programma (e agli utenti) per riconoscere i test singoli attualmente disponibili;
- *reports*: qui sono salvati i report in formato testo;
- *testTXT*: in questa cartella ci sono i file test in formato testo. Ognuno di essi rappresenta un Macro test da assemblare sulla base delle direttive lette da file. Il verificatore, nel preparare un nuovo test, si concentrerà prevalentemente su questi file.
- *testXML*: gli oggetti Macro_test serializzati, insieme ai propri report, vengono salvati in questa directory in formato XML. L'estensione dei file XML è ".tst'' ed il verificatore può leggerli ed eventualmente editarli

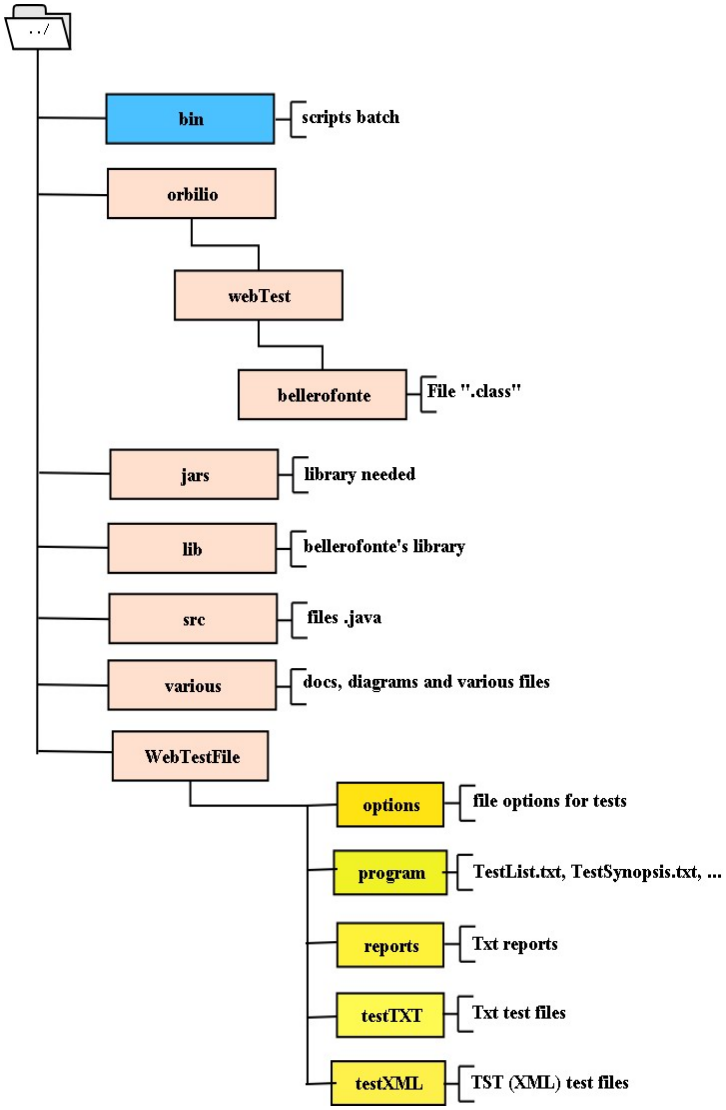

Figura 6.1: Directory di lavoro di Bellerofonte

con qualunque editor di testo o browser. La differenza tra i file test ".txt'' e quelli ".tst'' verrà illustrata tra breve.

Oltre alla directory webTestFile, all'interno del pacchetto ci sono altre directory di cui Bellerofonte non ha stretto bisogno ma che lo rendono più usabile:

- **bin**: contiene degli script batch che ne agevolano l'uso e il mantenimento.

  1. "belJAR.bat'', quando viene avviato, va ad eseguire l'archivio jar contenente Bellerofonte. Questo script può essere invocato sia da ambiente DOS/Windows che da Unix (cambiandone i permessi di esecuzione), passandogli o meno i parametri che richiede Bellerofonte. Provvede infatti lo stesso Bellerofonte ad avviare un processo interattivo che interroga l'utente sugli input mancanti. Per questa particolarità, belJar è adatto ad essere richiamato anche da un ambiente a finestre, semplicemente cliccandoci sopra;
  2. "makeJar.bat'' ha una funzione accessoria, utile agli sviluppatori di nuovi test singoli: esso aggiorna automaticamente l'archivio eseguibile "bellerofonte.jar'' con tutte le nuove definizioni ".class'' trovate. Per far questo va a leggere il file "manifest.mf'', dove tra l'altro viene impostata la variabile di ambiente "Classpath'' affinché Bellerofonte sappia dove trovare le varie librerie jar usate dai test singoli.

- **jars**: questa directory contiene gli archivi jar compressi con i pacchetti utilizzati nei Single test o da Bellerofonte;

- **lib**: dentro lib si trova il jar eseguibile "bellerofonte.jar'' ed il relativo file "manifest.mf'';

- **orbilio\webTest\bellerofonte**: qui viene aggiornato l'output della compilazione dei file sorgenti. Vi sono i file ".class'' delle classi di programma e dei vari test singoli. Bellerofonte controlla in questa directory la presenza delle definizioni dei test singoli prima di istanziarne gli oggetti;

- **src**: in src stanno i file ".java'' ovvero i sorgenti di Bellerofonte e dei Single test;

- **various**: in various sono stati inseriti i file di documentazione insieme ad immagini significative (come quelle che appaiono in questa tesi). In

particolare sono presenti (in vari formati) documenti per l'utilizzo e l'estensione di Bellerofonte e le sue API in formato HTML.

6.2 Eseguire Bellerofonte

Illustrato l'ambiente nel quale ci si dovrà muovere, si può procedere col descrivere come eseguire Bellerofonte, soffermandosi su come accedere alle sue funzionalità e senza entrare nuovamente nei dettagli della loro implementazione. Bellerofonte ha bisogno di alcuni parametri in input per essere eseguito:

- il nome di un file opzioni;
- il nome di un file test;
- una parola chiave che lo istruisca su come trattare il report ed il Macro test eseguito.

Come già si è detto nel capitolo precedente, i primi due valori sono necessari mentre il terzo è opzionale. Spostandosi nella cartella bin, la sintassi di esecuzione dalla riga di comando è la seguente:

```
belJar <OPZIONI.txt> [<TEST.txt>|<TEST.tst>]
(saveToMacroTest|compareToOld|compareToOldAndSave|<REPORT.txt>|<NULL>)
```

oppure semplicemente

```
belJar
```

In ogni caso è necessario preparare i file di cui il software fa richiesta prima dell'esecuzione. Bellerofonte li andrà a ricercare solo ed esclusivamente nelle directory predefinite.

6.2.1 Preparare il file opzioni

Il primo parametro obbligatorio da preparare è il file di opzioni. Questi file possono essere editati dall'utente nella directory webTestFiles\options con un qualsiasi editor di testo. La forma che deve avere un file opzioni è quella tipica dei *file Properties* di Java, in cui si ha una serie di coppie *chiave* = *valore*. Le scelte selezionabili in ognuno di questi file sono numerose e suddivise in tre categorie:

- Target options: descrivono il *target* del test, ovvero l'indirizzo dal quale il test dovrà partire;
- Client options: definiscono l'identità e le funzionalità generali del client, nella fattispecie Bellerofonte stesso;
- HTTPUnit options: sono più strettamente legate alle modalità di rappresentazione degli oggetti web incontrati da parte di HTTPUnit.

La maggioranza delle scelte non ha bisogno di modifiche da parte del verificatore (se non per particolari tipi di test), ma è ovviamente obbligatorio inserire almeno un target raggiungibile. Un esempio di file opzioni "pulito" (cioè non relativo a nessun sito testato) è "options.txt'', che si consiglia di utilizzare come modello (salvandolo quindi con un nuovo nome). Si è scelto di imporre la definizione separata dei file opzioni rispetto ai Macro test in quanto si suppone che il verificatore, oltre a crearsi una propria libreria di file test, realizzi anche un certo numero di opzioni relative ai siti da verificare. Ognuno di questi file sarà poi indipendente e combinabile liberamente con i file test.

Dato che l'attuale versione di HTTPUnit ha mostrato problemi ad interpretare alcuni linguaggi di scripting, una soluzione è quella di disabilitare, quando necessario, il parsing di tali linguaggi. Per ulteriori informazioni e aggiornamenti su questa lacuna di HTTPUnit si rimanda al suo sito web.

6.2.2 Preparare il file test

La fase successiva è quella di definizione del Macro test in un file test. I file test possono avere due forme e due ruoli diversi:

- Nel primo caso si suppone che il verificatore debba comporre un nuovo Macro test. Un nuovo Macro test viene descritto a partire da un file di testo in cui si elencano ordinatamente i nomi ed i parametri di ogni test singolo, separati dal carattere '|'. La sintassi di ogni riga di un file test è quindi

```
<NOME SINGLE TEST>|*(PARAMETRO N-ESIMO|)
```

L'elenco dei test singoli con la loro sintassi e la semantica dei parametri è inserito nei file "TestSynopsis.txt'' e "TestSynopsis.doc'' (presente anche in formato pdf). Il verificatore può trovare degli esempi e prenderli come punto di partenza per i propri test in webTestFiles\testTXT.

La definizione di un Macro test tramite file di testo implica la creazione di un nuovo oggetto Macro_test all'interno di Bellerofonte. Questa osservazione è importante quando si devono gestire i salvataggi: se partendo da un file di testo era già stato salvato un oggetto Macro_test in precedenza e questo conteneva dei report relativi a passate esecuzioni, l'aver forzato Bellerofonte a ricostruire un nuovo (ma omonimo) Macro_test provoca una sovrascrittura, con conseguente perdita dei report già salvati.

Nell'editare un file di test ci si deve ricordare che quel che si scrive è *case sensitive*, ovvero Bellerofonte è sensibile alla differenza tra maiuscole e minuscole. Riguardo ai parametri che seguono il nome c'è da dire che essi vengono di solito trattati tutti come stringhe e solo i Single test che ne fanno un uso particolare li convertono, dove serve, in valori numerici. I parametri, essendo delle stringhe, comprendono ogni carattere tra due delimitatori | (spazi inclusi). Bisogna inoltre stare attenti a non lasciare righe vuote nel file test, che Bellerofonte potrebbe tradurre in modo imprevedibile. Un file test vuoto invece corrisponde ad un Macro test nullo.

Un file test ha pressappoco la funzione del test script, ovvero ordina e definisce i passi di un test riferito ad una singola caratteristica del sito. Tutti i test singoli che sono inseriti in un file test dovrebbero quindi concorrere alla verifica di un unico aspetto. Per questo se un test singolo effettua una modifica sull'oggetto che testa, influisce sulle esecuzioni dei test seguenti.

Un semplice esempio di file test in formato testo è il seguente:

```
FollowLinks|string|Home|
CheckTitle|My HomePage|
```

- L'alternativa, citata in precedenza, alla definizione di un Macro test con un file di testo è quella di riutilizzare un oggetto XML serializzato, contenente un Macro_test già definito. In pratica la serializzazione comprende il salvataggio dello stato dell'oggetto Macro_test al termine della sua esecuzione: si memorizza il vettore dei Single test già costruito, il report attuale e le opzioni inizializzate (benché queste saranno comunque rilette e sovrascritte in successive esecuzioni). Un Macro test serializzato costituisce quindi un oggetto pronto per essere rieseguito, con in più la capacità di memorizzare e comparare i risultati delle sue esecuzioni. Altri due vantaggi dell'utilizzo di un file ".tst'', sono la maggior rapidità di esecuzione e la possibilità di visualizzare e modificare oggetti serializzati, perché codificati nell'ormai universale XML. Si deve però ricordare che un verificatore non può direttamente definire un oggetto di questo tipo. Esso deve necessariamente passare attraverso un Macro test su file di testo, il quale potrà essere serializzato in seguito alla sua prima esecuzione. Esempi di file ".tst'' si trovano nella cartella webTestFiles\testXML.

6.2.3 Gestire i risultati

Il terzo aspetto, opzionale ma propedeutico all'esecuzione di un Macro test, è quello che riguarda la gestione dei risultati, ovvero del report e dell'oggetto Macro_test. A seconda di quale parola chiave compaia come terzo parametro, Bellerofonte agisce di conseguenza:

- nel caso in cui nessuna stringa segua i nomi dei file opzioni e test, il programma semplicemente termina subito dopo aver eseguito il Macro test specificato (restituendo cioè solamente i risultati a video);
- quando compare qualsiasi stringa diversa dalle parole chiave riconosciute, il software la interpreta come nome di un file di testo. Tale file sarà

usato per salvare il contenuto del report attuale nell'apposita directory webTestFile\reports;

- inserendo la parola chiave *"saveToMacroTest"* Bellerofonte serializzerà il Macro_test ed il report attuali in un file XML con estensione ".tst'' nella directory webTestFiles\testXML. "saveToMacroTest" può produrre effetti molto differenti:

  1. se il Macro test viene letto da un file di testo e già esisteva un oggetto serializzato, questo viene sovrascritto (con conseguente cancellazione dei report contenuti);
  2. altrimenti se viene letto da un file ".tst'', "saveToMacroTest" provoca un salvataggio del report attuale nello stesso file XML. Un oggetto ".tst'' può contenere al massimo gli ultimi due report realizzati. Il salvataggio di un terzo report coincide con la cancellazione di quello più vecchio;

- la parola chiave *"compareToOld"* non implica il salvataggio del report attuale ma il confronto a video con i report precedenti. Questo parametro viene preso in considerazione da Bellerofonte solo se associato ad un Macro test già eseguito (quindi contenuto in un file ".tst''). Attualmente non è previsto il salvataggio su file delle schermate di comparazione;

- infine *"compareToOldAndSave"* provoca l'ovvia somma degli effetti delle due parole chiave precedenti. Si provvede a salvare il nuovo report sull'oggetto XML serializzato ed al tempo stesso si restituisce a video la sua comparazione con i due ultimi report salvati. Anche questa variante presuppone la lettura del Macro test da un file ".tst''.

In ogni caso il programma accetta i parametri solo nell'ordine stabilito e le parole chiave illustrate in questa sezione si escludono a vicenda.

6.2.4 La console interattiva

La console interattiva consente al verificatore di avere sotto controllo tutti i file (opzioni, ".txt'' e ".tst'') fino a quel momento creati. Infatti Bellerofonte, nel richiedere i parametri di input, presenta a video un elenco degli oggetti contenuti nelle directory delle opzioni e dei test, così che l'utente possa selezionarli semplicemente digitandone il numero progressivo. Qualora questo numero non fosse inserito oppure non corrispondesse a nessun file, il programma terminerebbe chiudendo la console e senza eseguire alcun test.

Il verificatore può rispondere alle richieste della console anche digitando per intero (cioè compresa l'estensione) il nome dei file. Invece, nella terza domanda relativa alle parole chiave, la stringa introdotta come nome di file report viene automaticamente completata con l'estensione ".txt'' quando questa sia stata omessa.

Se per mancanza degli input dalla riga di comando Bellerofonte è costretto ad avviare la console interattiva, la terminazione del programma dovrà essere preceduta dalla pressione del tasto "Invio''.

6.2.5 Esempi pratici

Per concludere questo capitolo sull'uso pratico di Bellerofonte, si riportano alcuni esempi:

1. **Verifica di un sito con parametri dalla riga di comando**:

   ```
   beljar orbilioLocal.txt testprova3.txt
   ```

   Output:

   ```
   C:\claudiosoft.bellerofonte\bin>echo off

   Bellerofonte: ver. 1.0 --- 2-ago-2003 11.41.12

   ------------------------------------------------------------
   REPORT OF TEST: testprova3.txt - 2-ago-2003 11.41.12
   ------------------------------------------------------------

   1.0] TEST: "SetFormContent" | Url: http://127.0.0.1/index.php
   ```

```
   Result expected: "Form: 0"
   Result expected: "Field: uname"
   Result expected: "Value to set: u1"
   + TEST PASSED: The field has been setted at the expected value

   # Seconds elapsed to accomplish the test: 0.10

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

2.0] TEST: "SetFormContent" | Url: http://127.0.0.1/index.php
   Result expected: "Form: 0"
   Result expected: "Field: pass"
   Result expected: "Value to set: u1"
   + TEST PASSED: The field has been setted at the expected value

   # Seconds elapsed to accomplish the test: 0.0

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

3.0] TEST: "SubmitForm" | Url: http://127.0.0.1/index.php
   Result expected: "Form: 0"
   Result expected: "Button: "
   + TEST PASSED: Button found and form submitted

   # Seconds elapsed to accomplish the test: 0.551

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

Total seconds of execution: 0.561


Result of tests in testprova3.txt:
  * 3 tests PASSED;
  * 0 tests FAILED;
  * 0 GENERAL ERRORS;
```

2. **Verifica di un sito con uso di console interattiva**:

   `beljar`

   Output:

```
C:\claudiosoft.bellerofonte\bin>echo off

Bellerofonte: ver. 1.0 --- 2-ago-2003 11.53.54

* Write the name of options file that will be used from list below:
```

```
0- options.txt
1- orbilioLocal.txt
2- orbilioThen.txt
3- otherLocal.txt
4- otherWWW.txt

* Write the name of test file that will be used from list below:

0- nullTest.txt
1- testButtonValue.txt
2- testFormField.txt
3- testImage.txt
4- testLink.txt
5- testProva.txt
6- testProva2.txt
7- testProva3.txt
8- testSubmit.txt
9- testTitle.txt
10- videoCall.txt
11- testLink.tst

* How must the report be treated ?
[1] "saveToMacroTest"
[2] "compareToOld"
[3] "compareToOldAndSave"
- nameReportInTXT
- null



-------------------------------------------------------
REPORT OF TEST: testFormField.txt - 2-ago-2003 11.54.03
-------------------------------------------------------


1.0] TEST: "CheckFormField" | Url: http://127.0.0.1/index.php
   Result expected: "Form: 0"
   Result expected: "Field: uname"
   + TEST PASSED: The form has the field expected

   # Seconds elapsed to accomplish the test: 0.10

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

Total seconds of execution: 0.10

Result of tests in testFormField.txt:
```

```
  * 1 tests PASSED;
  * 0 tests FAILED;
  * 0 GENERAL ERRORS;

(press Enter) --- 2-ago-2003 11.54.06
```

3. **Comparazione tra due report di un Macro test già serializzato in precedenza**:

   `beljar orbiliolocal.txt testFormField.tst compareToOld`

   Output:

```
C:\claudiosoft.bellerofonte\bin>echo off

Bellerofonte: ver. 1.0 --- 2-ago-2003 14.14.04

_____________________________________________
REPORT OF TEST: testFormField.txt - 2-ago-2003 14.14.06
_____________________________________________

1.0] TEST: "CheckFormField" | Url: http://127.0.0.1/index.php
   Result expected: "Form: 0"
   Result expected: "Field: uname"
   + TEST PASSED: The form has the field expected

 # Seconds elapsed to accomplish the test: 0.0

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

Total seconds of execution: 0.10

COMPARISON BETWEEN LAST TESTS (Actual Test , Old Test , Very Old Test)
_____________________________________________
CheckFormField , CheckFormField
- - - - - - - - - - - - - - - - - - - - - - - - -
PASSED , PASSED
- - - - - - - - - - - - - - - - - - - - - - - - -
0.20 , 0.10
_____________________________________________

Result of tests in testFormField.txt:
  * 1 tests PASSED;
  * 0 tests FAILED;
  * 0 GENERAL ERRORS;
```

4. **Applicazione di Bellerofonte in vari test consecutivi, a formare una test suite**:

   ```
   beljar orbiliolocal.txt testprova2.txt saveToMacroTest |
   beljar orbiliolocal.txt testprova3.txt rep2.txt
   ```

   In questo caso il risultato è la visualizzazione a schermo dell'ultimo report compilato ("rep2.txt''), il suo salvataggio come file di testo nella cartella webTestFiles\reports e la serializzazione del file di test "testprova2.txt'' nella directory webTestFiles\testXML;

Parte III

La piattaforma Orbilio: sviluppo e testing

Capitolo 7

Breve descrizione di Orbilio

In questo capitolo verrà descritta sommariamente l'applicazione web per la quale Bellerofonte è stato ideato e realizzato: Orbilio.

Orbilio nasce e si sviluppa all'interno del Dipartimento di Sistemi ed Informatica dell'Università di Firenze ed è attualmente seguito da un apposito gruppo di lavoro formato da professori e studenti.

Esso si propone come strumento di supporto alla didattica e, andando oltre, come piattaforma di *e-learning*, in grado di offrire funzionalità per l'interazione in tempo reale. Prima di Orbilio, il sito web del Dipartimento (all'URL **if.dsi.unifi.it**) già gestiva una serie di servizi tesi a raccogliere informazioni utili per i professori e fornire un riferimento ai vari corsi per gli studenti. C'era però l'esigenza di uniformare questi servizi in una piattaforma definitiva, che offrisse, per ogni corso universitario, una comune veste grafica ed un'omogenea implementazione funzionale.

Dopo aver ispezionato varie alternative presenti attualmente nel ramo del supporto alla didattica, è stata scelta come piattaforma di partenza *Claroline*, un'applicazione realizzata all'Università Cattolica di Lovanio, in Belgio. Claroline presenta alcune evidenti qualità:

1. è un progetto con licenza Open Source, ovvero permette la modifica e la successiva redistribuzione dei file sorgenti;

2. è un software sviluppato sulla base di tre tecnologie molto diffuse e

conosciute: il linguaggio PHP, il server Apache ed una base di dati MySQL;

3. presenta già tutta una serie di servizi di base disponibili ai professori ed agli studenti: una *home page* separata e personalizzabile per ogni corso, la gestione di gruppi di lavoro, forum e così via.

Dopo un certo periodo di prova sul sito del Dipartimento sono però emersi anche i limiti e le lacune di questo software:

- benché supporti molte lingue diverse nell'interfaccia, la gestione delle traduzioni non è risultata abbastanza precisa;
- il codice sorgente appare caotico e dunque difficile da modificare. Allo sviluppo di Claroline devono aver preso parte, in momenti differenti, persone di varia lingua. Il risultato è una certa ambiguità nelle definizioni, dovuta prevalentemente alle discrepanze linguistiche e stilistiche in commenti, nomi di variabile, campi del database ed altro ancora;
- le modalità di gestione del database subiscono frequenti modifiche. Ciò si ripercuote direttamente sulla complessità degli aggiornamenti da una versione all'altra;
- gli aspetti amministrativi della piattaforma nel suo complesso non sono sufficientemente flessibili e semplici. Non dispongono inoltre di nessun meccanismo in grado di evitare involontari interventi manuali dannosi.

Sulla base di queste constatazioni è stato deciso dal gruppo di lavoro sopra menzionato di effettuare un *fork* dalla piattaforma Claroline ad una nuova piattaforma: Orbilio. Le modifiche da apportare erano in effetti troppe per essere gestite sempre all'interno del progetto Claroline. Inoltre, alla base di Orbilio, sta la volontà di offrire una vera piattaforma di e-learning, cosa che Claroline non mirava ad essere.

Così in Orbilio sono stati migliorati o aggiunti i seguenti aspetti:

- eliminazione di alcuni file non più utilizzati in Claroline, ma comunque compresi nel pacchetto. Il risultato è una maggiore snellezza del progetto;

- introduzione di nuove componenti. L'innovazione più importante, primo passo verso la realizzazione di una piattaforma di e-learning, è stata l'aggiunta della funzionalità di *video-ricevimento*, in grado di fornire un contatto diretto tra professore e studenti tramite *web-cam*;

- integrata una nuova *procedura di autenticazione*[18], capace di rendere effettiva la corrispondenza tra i reali studenti del corso di laurea e gli studenti registrati ad Orbilio nonché di semplificare le operazioni di gestione;

- migliorato l'*aspetto amministrativo*, con aggiunta di nuove schermate di configurazione guidata e gestione assistita del database.

Orbilio punta inoltre ad essere maggiormente usabile ed installabile anche da utenti con scarse conoscenze informatiche. Questo per favorire la sua diffusione anche al di fuori del Dipartimento. Orbilio rimane comunque un progetto Open Source. Per informazioni più approfondite sulle piattaforme di e-learning in generale e su Orbilio in particolare si vedano i documenti [16] e [17] citati in bibliografia.

Capitolo 8

Testing di Orbilio con Bellerofonte

Parallelo al progetto Orbilio descritto nel capitolo precedente, si è sviluppato Bellerofonte. Essendo Orbilio nato da oggettive necessità di modificare aspetti lacunosi della piattaforma Claroline, si è resa subito palese l'esigenza di garantire uno standard di qualità minimo nelle modifiche introdotte. Non si voleva infatti ripetere l'errore fatto in Claroline, dove numerose aggiunte di funzionalità incoerenti tra loro hanno portato ad una confusione diffusa a tutti i livelli ed a un'usabilità ridotta.

Bellerofonte è quindi rivolto specialmente ai futuri sviluppatori di Orbilio, i quali, ampliando le funzionalità della piattaforma con nuovi servizi, saranno chiamati a garantirne la correttezza tramite appositi test.

In questa prima versione di Bellerofonte è stata presentata una serie di test singoli i quali, una volta combinati tra loro, possono fornire un'adeguata gamma di test funzionali, regressivi, di caricamento e di interfaccia. Le modalità con cui realizzare tali test sono quelle illustrate nel capitolo 6.

Sempre nell'ambito di questa tesi è stato realizzato un semplice test che verifica la raggiungibilità di un certo numero di file di Orbilio, una volta terminata la procedura di installazione. Questo test, basato sul test singolo

"ReachAll'', può dunque considerarsi un primo esempio di Installation test.

8.1 Un semplice esempio di Installation test

Orbilio punta ad essere una piattaforma *user-friendly* non solo dal lato dell'utente, ma anche da quello dell'amministratore. É una qualità fondamentale di quei software che ambiscono ad uscire dall'ambiente in cui sono stati creati per essere installati ed usati anche da persone non addette. Per questo motivo, tra i vari test che potevano essere realizzati, se ne è privilegiato uno post-installazione, che consentisse ad un amministratore di controllare automaticamente lo stato della piattaforma appena installata (senza però includere obbligatoriamente questa verifica in Orbilio). Al tempo stesso si è voluta mostrare la versatilità di Bellerofonte e la sua capacità di integrarsi con interfacce grafiche esterne. Il risultato ottenuto conferma queste proprietà, ma, come lo stesso Bellerofonte, non punta ad essere definitivo e rimane suscettibile alle future modifiche di Orbilio.

In questo paragrafo non si scenderà in nessun dettaglio tecnico circa la realizzazione dell'interfaccia grafica che coordina la raccolta dei dati e l'esecuzione del test. Ciò perché questa particolare implementazione non è rilevante per il web testing e non è neppure vincolante ai fini di future interfacce grafiche di Bellerofonte (la già citata Pegaso). La scelta, in questo caso, è caduta su *Swing* di Java per vari motivi:

- si integra perfettamente con Bellerofonte sotto molti aspetti: quello tecnico e quello della portabilità in primo luogo;
- c'è un'ampia gamma di componenti e di strumenti per il disegno di interfacce già pronti;
- consente di creare un'interfaccia *stand-alone*, che cioè ha bisogno solo della Java Virtual Machine per funzionare.

I requisiti quindi coincidono con quelli di Bellerofonte.

L'interfaccia realizzata per l'installation test, che si presenta una volta lanciato il file batch "TestGUI.bat'', è mostrata in figura 8.1

L'installation test presume, ovviamente, la disponibilità di Bellerofonte e di Orbilio. É dunque mostrato a chi esegue il test un form nel quale dovranno essere inseriti i seguenti dati:

1. l'URL di Orbilio, ovvero della directory dove è collocato il suo "index.php'';
2. il path assoluto della root di Bellerofonte, cioè il percorso nel file system che porta alla directory "claudiosoft.bellerofonte'';
3. l'identificatore di login e la password per accedere ad Orbilio come amministratore.

Al di sotto dei campi citati c'è un'area di testo con i file standard di Orbilio (versione 1.0) che Bellerofonte tenterà di raggiungere nel suo test. A questo elenco di file l'amministratore (o più facilmente lo sviluppatore) di Orbilio può aggiungerne altri. Se si volesse rendere permanente questa selezione si dovrebbe invece intervenire sul file di testo "listFile.txt'' contenuto nella directory "files'' della directory "claudiosoft.testGUI''.

L'interfaccia raccoglie tutti i dati (effettuando anche delle verifiche sulla loro correttezza) e, dopo la pressione del bottone "Start'', li rielabora: crea il file di opzioni, i file (usati nei test "ReachAll'') contenenti gli URL delle pagine da raggiungere ed il file con l'elenco dei test singoli (in formato testo).

Tutti i file creati, a parte il report, sono temporanei e vengono cancellati all'uscita dal test.

Al termine della fase preparatoria, se non sono state incontrate difficoltà, viene lanciata l'esecuzione di Bellerofonte in un processo separato. L'output di esecuzione scorrerà nella shell mentre il report verrà visualizzato in una finestra *pop-up* aperta appositamente. In base al valore di uscita del sotto-processo l'interfaccia si aggiornerà di conseguenza, confermando o smentendo al verificatore il buon funzionamento di Orbilio.

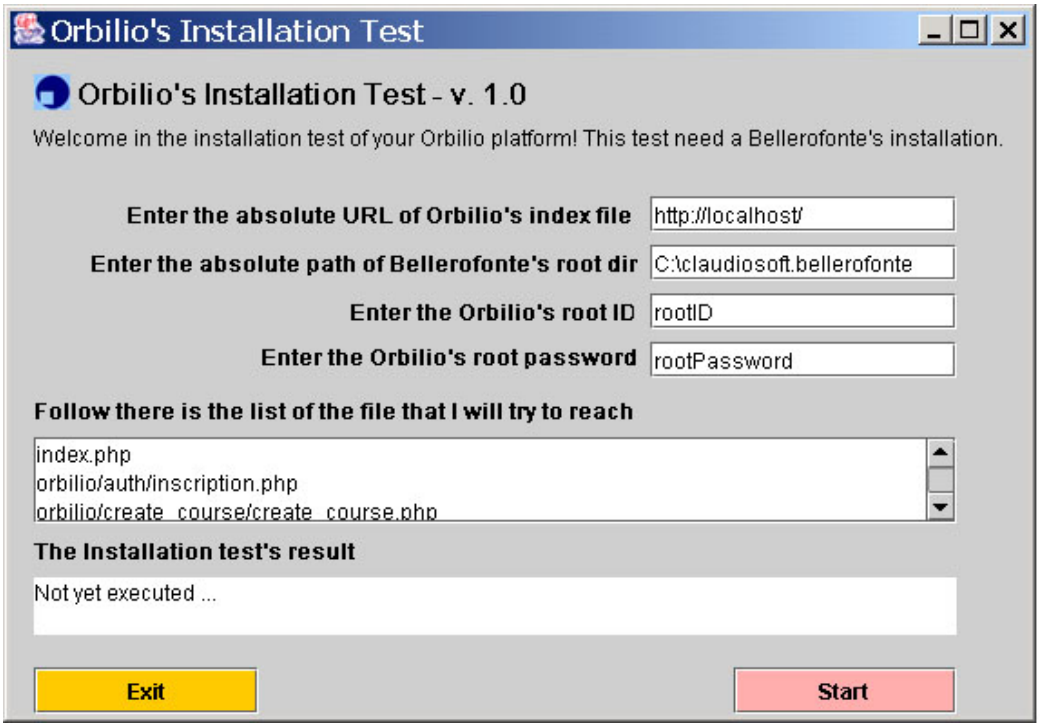

Figura 8.1: Interfaccia dell'Installation test

Il test assemblato dell'interfaccia non fornisce in realtà una prova "forte'' della corretta installazione di Orbilio: esso è costituito da due esecuzioni separate del test singolo "ReachAll''; la prima è volta a verificare le pagine di base mentre la seconda viene lanciata all'interno dell'area di amministrazione (per questo servono identificatore e password di amministratore). Dato che il test non interessa le funzionalità ma solo la raggiungibilità delle pagine elencate, ci si deve accontentare di dedurre che ogni link è funzionante e dunque presumibilmente il web server e il database sono ben configurati. Del resto si può ipotizzare che la verifica delle singole componenti sia stata già fatta dagli sviluppatori con appositi test funzionali e di usabilità.

L'interfaccia di questo installation test risulta comunque piuttosto spartana, mancando di facilitazioni quali la possibilità di salvare i report e di selezionare i path con rappresentazioni ad albero del file system. Se si considera però che altri tipi di test potrebbero essere facilmente assemblati sfruttando questa implementazione (con al più poche modifiche), si ha la riprova che Bellerofonte è sufficientemente versatile da essere usato sia tramite *GUI* che tramite riga di comando.

8.2 Conclusioni

Vi sono ormai molti libri e articoli che riguardano gli argomenti trattati in questa tesi. Malgrado ciò, il testing ed in particolare il web testing rimangono attività sconosciute o marginali tra la maggior parte dei programmatori. Se da una parte è vero che non esiste una metodologia definitiva e completa da applicare, è anche vero che l'arte (o l'intuizione) di evitare errori non viene coltivata in modo sistematico e "scientifico'', dando maggior spazio ad altre problematiche. Nel corso di questa tesi si sono illustrati ad alto livello i nodi che gli sviluppatori, i verificatori e gli utenti si trovano a dover sciogliere quando incontrano un bug e quali vantaggi ne possono derivare. Si è poi presentato uno dei tanti possibili strumenti a disposizione per "sbrogliare'' questa matassa. La speranza è che lo sviluppo di Orbilio e di Bellerofonte

proceda parallelamente, così da garantire correttezza e affidabilità, ma anche una maggiore sensibilità verso la qualità del proprio software.

Bibliografia

Articoli internet

[1] Testing Web Services
Neil Davidson
www.webservices.org

[2] Web services: un approccio morbido
Luca Balzerani
www.latoserver.it

[3] Web Services
Elena Bernardi
www.evectors.it

[4] Web Services
Massimiliano Bigatti
www.mokabyte.it

[5] Il mistero dei web service... svelato
www.idg.it/networking

[6] What is software testing? And why is it so hard?
James Whittaker
Florida Institute of Technology

[7] Software debugging, testing and verification
B. Hailpern / P. Santhanam
IBM System Journal, vol. 41, n. 1 2002

[8] eXtreme Programming Methodology
www.extremeprogramming.org

[9] Quality First - *Mary Hayes*
www.informationweek.com

Testi

[10] Programming Web Services with SOAP
James Snell
O'REILLY

[11] Testing applications on the web
Hung Q. Nguyen
WILEY

[12] Computer Networks and Internets with Internet Applications - third edition
Douglas E. Comer

[13] Reti di calcolatori
Larry Peterson / Bruce Davie
Zanichelli

Relazioni, Tesi ed altre pubblicazioni

[14] Il computer che si ripara da solo
Armando Fox / David Patterson
Le Scienze n. 419 Luglio 2003

[15] Servizi remoti nei sistemi distribuiti: RPC, thread e panoramica su Corba
Bracciali / Farini / Tortorelli

[16] Orbilio: una piattaforma Open Source per l'e-learning
Antonio Talarico
Università degli Studi di Firenze, Facoltà di SMFN, CDL Scienze dell'Informazione

[17] E-learning e Videoconferenza: Teoria ed Applicazione
Dario Di Minno
Università degli Studi di Firenze, Facoltà di SMFN, CDL Informatica

[18] L'autenticazione e la gestione delle frequenze nella piattaforma Orbilio
Federico Chicchi
Università degli Studi di Firenze, Facoltà di SMFN, CDL Informatica

Indice analitico

Acceptance test, 24
Analisi dinamica, 13
Analisi mutazionale, 13
Analisi statica, 13

Black-Box testing, 14
Browser, 31
Bug, 14

Check list, 23
Claroline, 82
Client, 29
Codice embedded, asserzioni, 21
Correttezza software, 13
Criterio di copertura, 19

Database distribuito, 44
Database relazionali, 43
Database server, 43
Debugging, 13
Dijkstra, tesi di, 12

Failure, 13
Fat client, 30
Fat server, 30
Fault, 13
Fixed, fissare errori, 14
Formalismi descrittivi, 20

Gray-Box testing, 14

HTML, 31
HTTP, 31

Integrazione dinamica, 34
Integrity error, 44
Interfacce col File System, 19
Interfacce di comunicazione, 19
Interfacce software, 19
Interfacce umane, 18
IP, 30

Macro test, 56
Metafora, 41
Millennium Bug, 17
Modello Peer, 35

Oracolo, 20
Output error, 44

Polimorfismo, 63
Problema della fermata, 12

Scenario di test, 19
Self-testing, 21
Server, 29
Servizio di rete, 32
Sever multi-livello, 30

Single test, 55
Successo nei test, 13

TCP, 29
Test case, 15
Test data adequacy criteria, 19
Test di regressione, 21
Test plan, 15
Test script, 15
Test suite, 15
Test types, 38
Testability, 23
Testing in grande, 15
Testing in piccolo, 14
Tool di testing automatici, 20

URL, 31

Validazione, 13
Verifica, 13

Web, 31
Web server, 31
Web service, 33
Web service architecture, 34
Web spider, 51
White-Box testing, 14

XML, 34