UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

Tesi di laurea specialistica

Progettazione di un tool per il test automatico di applicazioni Java dotate di interfacce grafiche

Anno Accademico 2010/2011

relatore
Ch.mo prof. Stefano Russo

correlatore
Ing. Roberto Pietrantuono
Ing. Gabriella Carrozza

candidato
Maurizio Sorrentino
matr. 885/472

Alla mia famiglia

Indice

Indice delle Figure

Indice delle Tabelle

Introduzione

Questo lavoro di tesi ha come obiettivo la progettazione e l'implementazione di un sistema a supporto al processo di *testing*.

Il lavoro elaborato può essere suddiviso in due fasi. La prima, svolta prevalentemente presso il laboratorio Nazionale del CINI di Napoli; la seconda, presso lo stabilimento di SELEX Sistemi Integrati sito in Giugliano in Campania (NA) e più precisamente presso il centro di ricerca del SESM.

Dopo una fase iniziale volta allo studio di documenti, ricerca ed analisi delle tecnologie esistenti, si è passati alla progettazione e all'implementazione di una prima *release* del *tool*. Tale strumento a supporto alle attività di Verifica e Validazione Software è stato denominato "Testeryx".

Il SESM è un centro di ricerca e sviluppo precompetitivo di aziende Finmeccanica, quali SELEX Sistemi Integrati e SELEX Galileo, attivo nel campo della difesa, gestione del traffico aereo e logistica di grandi apparati.

Opera su diversi livelli che vanno dalla ricerca di base alla ricerca industriale, allo sviluppo precompetitivo, alla formazione di risorse per le aziende consorziate. È molto attivo nella promozione e gestione di progetti di ricerca cofinanziati dalla Commissione Europea, dal MUR (Ministero Università e Ricerca) e dalle Regioni.

Particolare interesse è stato sempre dedicato allo sviluppo di soluzioni prototipali per problematiche legate al trasporto aereo, in particolar modo, le attività di testing fondamentali per poter offrire ai clienti prodotti con qualità superiori rispetto a quelli offerti dalla concorrenza.

Le funzionalità da sviluppare, offerte dal sistema, sono le seguenti:

- supporto all'automatizzazione di test per applicazioni basate su interfacce grafiche;
- stesura della documentazione a corredo, quali STD (*Software Testing Description*) e STR (*Software Testing Report*), secondo lo standard MIL-STD-498;
- formalizzazione di linee guida da seguire per la stesura di casi di test al fine di facilitarne la loro generazione automatica e successiva generazione di script di test automatici.

Il presente elaborato di tesi è strutturato nel seguente modo.

Nel primo capitolo sono introdotti i concetti legati alla fase di *V&V* nell'ottica dell'automatizzazione dell'attività di *testing* in generale.

Nel secondo capitolo sono riportate tutte le tecnologie e metodologie atte a collaudare sistemi interattivi dotati di interfaccie grafiche; sono, inoltre, analizzati gli strumenti a supporto del *testing* funzionale, risultato della ricerca effettuata per esaminare l'attuale stato dell'arte.

Nel terzo capitolo viene mostrata la progettazione del software da dover realizzare, di cui è stata rilasciata la versione *beta*.

Infine, la tesi si conclude con il quarto capitolo, nel quale si descrive un caso di studio sperimentale; sono, quindi, presentati i risultati ottenuti e le considerazioni scaturite.

Capitolo 1

Il testing

Nell'attuale scenario dello sviluppo di sistemi software da impiegare in scenari critici si sta ponendo particolare attenzione ed importanza ad una delle fasi più delicate e costose: il *testing*. Da studi fatti a riguardo è stato dimostrato che la gran parte dei costi totali di sviluppo (fino a più del 50%) è dovuta al processo di verifica, soprattutto in sistemi complessi e di grandi dimensioni. Alcune ricerche hanno mostrato quanto gli investimenti nella fase di test portino a guadagni significativi. Diversi studi condotti evidenziano come il vantaggio ottenuto dall'organizzazione possa andare dal 20% fino addirittura al 70% del costo totale di produzione. Il vantaggio di investire nel test dipende poi dalla capacità dell'organizzazione di mettere in piedi, nelle prime fasi del ciclo di vita, politiche di lungo termine, che prevedano anche sforzi iniziali di pianificazione, laddove ad esempio il test è considerato un'attività che inizia solo dopo l'implementazione. In letteratura, è comunemente accettato, anche se non molto praticato in ambito industriale, il principio secondo cui, rilevare problemi e guasti nelle prime fasi del ciclo di vita porta a notevoli risparmi. Nello studio condotto da *Neal et al.* gli autori riportano un caso di studio sull'efficienza ed i costi della *Independent V&V*[1] di un prodotto della NASA, evidenziando, e confer-

[1] Il V&V (Verifica e Validazione) è un processo che punta a mostrare che il software sia conforme alle specifiche e che soddisfi le aspettative del cliente.

mando, che l'efficienza della *V&V* è maggiore quando essa viene applicata sin dalle prime fasi del ciclo di sviluppo. Questo fondamentale principio è mostrato già dai primi lavori sulla *V&V*. Tali studi evidenziano, inoltre, la capacità della *V&V* di rimuovere guasti nelle prime fasi del ciclo di vita; ciò evidentemente oltre a ridurre i costi di sviluppo impatta positivamente anche sui costi di manutenzione. La riduzione dei costi di sviluppo passa necessariamente attraverso la riduzione dei costi di *V&V*.

1.1 Cos'è il *testing* e a cosa serve

I programmi possono contenere errori. Un test consiste nell'esecuzione di un frammento di codice per verificare che funzioni. Una definizione più rigorosa afferma che:

"Il *Testing* è l'attività di specificare, designare ed eseguire test".

La normativa IEEE 610 definisce il *testing* come:

"Un'attività nella quale un sistema o le sue componenti sono analizzate sotto particolari condizioni, al fine di osservarne e registrarne il comportamento, valutando i differenti aspetti".

L'attività di test è una tecnica che è passata negli ultimi 20 anni dalla semplice intuizione alla scienza, dall'analisi personale ad una pratica ben consolidata, mossa dalla teoria e dall'esperienza di chi la compie. Tale attività, è comunque più complessa di quanto non sembri, ed è necessario capirne l'importanza e le sue modalità di attuazione, troppo spesso valutate ed eseguite in maniera del tutto soggettiva, senza alcuna metodologia consolidata, e senza nessun modello di riferimento [1].

1.2 L'importanza del *testing* nella realizzazione del Software

Un'analisi attenta delle imperfezioni del codice e dei suoi scostamenti da quanto progettato è un passo essenziale per la realizzazione di un prodotto valido, fun-

zionale, che soddisfi la clientela e non vanifichi l'attività di realizzazione del prodotto. Come andrebbero realmente le cose se la società distribuisse un prodotto senza prima verificarne la funzionalità e l'efficienza, dipende profondamente dalla tipologia del software, dalla sua complessità, dal pubblico finale e da altri fattori, ma nella quasi totalità dei casi la mancanza di test porta inevitabilmente ad un prodotto ancora "progetto", non finito, ed incompleto. Realtà di questo tipo si sono sempre avute nella storia del commercio informatico: società mosse dal fervore di superare la concorrenza hanno finito per mandare in fumo oltre che risorse (denaro/uomini) anche credibilità acquisite in lunghi tempi.

Esempi opposti arrivano invece da aziende di "affidabilità" consolidata, che producono software per i quali ogni difetto potenziale costituisce una perdita d'immagine e di *leadership* di mercato: un facile esempio è quello del campo dei *Data Base Management System*, nel quale aziende come *IBM* o l'*Oracle* puntano su prodotti di qualità, rilasciando nuove *release* del prodotto, basandosi su un attento e lungo test. La ragione, in questo caso, è anche dettata dal fatto che prodotti come i *DBMS* vengono utilizzati per la gestione di grosse quantità di dati di imprese di vario genere, contenenti informazioni riservate, importanti e di alto valore economico: ogni perdita di dati costituisce una perdita pesante in termini di denaro che si ripercuote sull'impresa stessa.

Attualmente ogni processo produttivo (su vasta scala in maniera particolare) è dotato di una fase di analisi ben definita, alla quale molte società dedicano anche il 35 – 40% circa del tempo complessivo di realizzazione del prodotto. Il tempo di completamento del prodotto, dovrebbe poter essere rispettato al fine di soddisfare la clientela, e protratto solo a fronte di gravi anomalie che devono essere assolutamente risolte [2].

Cosa cercare

È importante capire qual è l'oggetto dell'attività di test: si tratta nella generalità dei casi, della ricerca di tutto ciò che costituisce una discrepanza tra quanto *ri-*

chiesto dal cliente e *quanto realizzato*. Questo perché, non sempre è possibile rispettare i requisiti fissati in progettazione, a causa di fattori per lo più implementativi. A questa analisi si aggiunge anche la ricerca di tutti quegli errori presenti nel prodotto che lo rendono non adatto ad un'immediata distribuzione: tali differenze debbono essere raccolte e comunicate agli sviluppatori che provvederanno ad una loro risoluzione. Nella realtà solo una parte dei problemi è costituita da errori; un'altra, invece, da problemi più gravi, non riconducibili ad anomalie o imprecisioni, quanto piuttosto a logiche contorte, impossibilità/difficoltà di adattamento ad ambienti ed evoluzioni, difficoltà di approccio ed utilizzo etc.. Fattori, tutti strettamente connessi alla qualità del prodotto. Il tester avrà il compito di individuarli, comunicarli ed eventualmente presentare suggerimenti per migliorare il prodotto.

Occorre premettere che il *test non sarà in grado di individuare quegli errori quiescenti*, che rimangono "nascosti" durante i test e che poi si manifestano quando il software viene utilizzato in particolari condizioni riproducibili dai soli utenti. In questo caso, solo gli utilizzatori finali potranno sostenerci in una completa analisi [4,5].

Come e quando operare

La ricerca delle anomalie presenti nel software è una pratica, che è stata condotta troppo spesso con criteri personali e talvolta senza alcuna metodologia: un'attività così organizzata e poco documentata è senza dubbio scarsamente produttiva, oltre che potenzialmente confusionaria e poco efficiente. È importante, così come si fa con un'applicazione, dedicare del tempo alla creazione della documentazione iniziale, che costituisce una linea guida per la realizzazione dell'attività di test. Una completa documentazione dovrebbe essere costituita da:

- Un piano di test (*Software Testing Plan Documentation*), nel quale viene descritta nei particolari l'intera attività di test;
- Una descrizione dei *test-case* (*Software Testing Description Documentation*),

nel quale viene indicato l'oggetto del test, i valori ricevuti in ingresso, il risultato atteso e quello riscontrato (ciascun oggetto del test avrà il suo test-case);

- Un report dei *test-case* (*Software Testing Report Documentation*), nel quale viene indicato il risultato del caso di test.

Un test-case è un particolare insieme di dati e procedure utilizzati per controllare il comportamento di un preciso aspetto del programma; sono progettati prima di eseguire fisicamente i test, ed un gruppo di test-case costituisce un test suite. La documentazione e l'insieme delle prove possono essere suddivise tra differenti tester al fine di minimizzare i tempi o intensificare l'attività di ricerca delle anomalie, riunendo, poi, tutta la documentazione creata in un unico piano di test finale.

La creazione di un'efficiente documentazione non è il solo problema che si pone nella pianificazione di un'attività di test. Anche la scelta del *momento* è un passo importante, considerando aspetti come il tempo effettivo rimanente, il numero di tester disponibili, la qualità che si desidera raggiungere; aspetti che non possono essere trascurati, e che condizionano direttamente l'attività di test modificandone la durata e le fasi. Inoltre, tester e sviluppatori hanno generalmente una visione differente delle cose. Il documento, oltre a costituire un punto di riferimento per le future attività dell'azienda, andrà in mano al cliente: questo dovrebbe far capire l'importanza di un elaborato preciso, chiaro, completo.

Il fondamento di una buona attività di test è sicuramente la scelta del momento nel quale iniziare la ricerca dei bug. Storicamente si sono diffusi principalmente due approcci: quello a *cascata*, che prevede l'esecuzione dei test a prodotto concluso, una volta che gli sviluppatori dispongono di una "beta" stabile del prodotto, e quello *iterativo incrementale*, che segue, invece, lo sviluppo del software. Il

concetto è fortemente legato all'*eXtreme Testing*[2]. Quale differenza esiste tra i due approcci? L'approccio a cascata è facile da mettere in pratica e da organizzare, ma i suoi limiti sono evidenti con grossi progetti: se la realizzazione di un prodotto portasse via dei mesi, fino al prodotto concluso non avremmo possibilità di verificare l'aderenza alle specifiche progettuali, ai requisiti dell'utente, ai requisiti di qualità, etc. Il rischio di un lavoro mal fatto e lontano dalle richieste dell'utente è alto.

Come alternativa all'approccio a cascata, quello iterativo incrementale prevede a priori una scomposizione del prodotti in unità; le unità vengono implementate singolarmente, testate ed inserite in opportune *release*, che essendo costituite da ridotti gruppi di funzionalità, sono più facilmente analizzabili. A man mano che la produzione del software va avanti il prodotto viene incrementato di nuove unità e testato nuovamente nel complesso. L'attività di test affianca così da subito l'attività di produzione, verificando l'aderenza agli standard, ai requisiti, agli obiettivi, etc.. Una migliore organizzazione e rispetto delle specifiche consente, anche, un migliore coinvolgimento del cliente, che può verificare l'attività dell'azienda alla quale ha commissionato il lavoro. Per contro, tale approccio richiede però un'attenta pianificazione iniziale che stabilisca tempi e contenuti delle *release* da esaminare e discutere con il cliente. È chiaro che vi deve essere anche una disponibilità da parte del cliente stesso di visionare periodicamente dette *release*, cosa non sempre possibile.

Possiamo schematizzare quanto detto in una tabella come segue, evidenziando pregi e difetti delle due metodologie:

[2] eXtreme Testing, introducendo i concetti di *Unit Test* e *Function Test*, prevede di creare un software modulare e scomponibile in unità. Per ciascuna unità, si creeranno degli *unit test*, che verificheranno la mancanza di errori. L'XT prevede di accettare l'unità solo quando la percentuale di successo del test è pari al 100%.

Tabella 1 Comparazione dell'approccio a Cascata e quello Iterativo incrementale.

	Cascata	Iterativo incrementale
Pregi	Facile da implementare	Permette di focalizzare l'attenzione su piccole porzioni di programma alla volta
	Facile da organizzare	Maggiore Facilità di test
		Maggiore aderenza alle specifiche ed ai requisiti
		Maggiore informazione al cliente
Difetti	Dinanzi ai progetti complessi diviene lungo e può richiedere diversi tester	Richiede la necessità di una attenta pianificazione
	Allontana il cliente dall'attività dell'azienda	Disponibilità immediate di tester
	Rischio di dimostrare troppo tarsi un prodotto non valido	Disponibilità del cliente

1.3 Il *Testing* automatico

L'attività del tester viene svolta quasi sempre manualmente; ci possono, però, essere dei casi nei quali sia più conveniente eseguire dei test automatizzati, utilizzando opportuni strumenti software in grado di sostituire il tester nella sua attività.

Le ragioni, che dettano questa necessità, sono riconducibili principalmente a motivi di tempo e di costi, ma raramente anche alla tipologia dell'applicazione: un esempio ci arriva dagli algoritmi genetici, nei quali la mole di calcoli da effettuare e la ripetitività delle operazioni spingono qualunque tester a desistere da una analisi manuale del prodotto. Progetti che richiedono un test intenso svolto manualmente da parecchi tester sono irreali e sicuramente anti-economici, quando lo stesso lavoro può essere eseguito in maniera efficiente da un software. In effetti, il test dovrebbe essere eseguito manualmente, in quanto così è possibile definire meglio il lavoro, organizzarlo ed eseguirlo in maniera più accurata. In al-

cuni casi, l'automazione ed il test manuale sono spesso intercambiabili, mentre in altri scenari è più opportuno scegliere l'uno o l'altro, valutando attentamente i costi ed i benefici di ciascuna tecnica. Tali caratteristiche sono state riassunte nella seguente tabella.

Tabella 2 Comparazione tra testing automatico e manuale.

	Test Manuale	Test Automatizzato
Pregi	Facilmente pianificabile	Basato su un software acquistabile
	Facilmente organizzabile	Può sostituire molti tester
	Accurato	È rapido e crea documentazione
Difetti	Può essere eccessivamente costoso (tempo/denaro)	Spesso impreciso
	Può essere eccessivamente lungo	Poco accurato
	Possono volerci molti tester	Poco Specifico
		Può essere costoso

Un problema che nasce spontaneamente, è quello riguardante la creazione della documentazione per il test e la pianificazione e realizzazione dei *test-case*. Nel caso di test manuale, il problema è relativo, in quanto il tester può eseguire sequenzialmente le prove e valutarne l'efficacia, eventualmente provvedendo ad una ristrutturazione o ad ulteriori analisi nel caso si accorgesse di anomalie. Nel caso di test automatizzato le cose si complicano, in quanto occorre non solo capire il funzionamento del software di test utilizzato, la sua copertura del codice, la sua affidabilità, ma anche, distinguere quali parti del prodotto si ritiene opportuno testare automaticamente e quali invece esaminare manualmente [3, 8, 12].

La migliore soluzione in questo senso è costituita da un buon compromesso tra due tecniche: *un'analisi sommaria ma completa da svolgersi sul prodotto e seguita da un affinamento manuale*, intensificando le prove in quei moduli del programma che hanno dimostrato minore robustezza o affidabilità. È quindi il

caso di dedicare una parte sostanziale del proprio lavoro alla redazione di un buon piano di test, che permetta di unire la velocità dell'automazione all'accuratezza dell'analisi manuale. Se, poi, sia conveniente optare per un test automatizzato e quanti *bug* potenzialmente non scoperti possano esistere, sono fattori che variano da progetto a progetto. Un secondo importante aspetto per l'azienda che esegue i test è quello di valutare se sia più opportuno creare personalmente un'applicazione di automatizzazione, oppure, acquistarla sul mercato. La scelta non è motivata solo da ragioni di costo, ma anche da motivi legati all'efficienza: i software acquistati da terze parti, hanno caratteristiche generali, per via del fatto di poter venire incontro alle molteplici esigenze della potenziale clientela; software personalizzabili sono possibili, ma occorre valutare i tempi, i costi e le modalità di attuazione. Valutiamo, però, la cosa più accuratamente: creare un'applicazione per testare un prodotto può essere, spesso, eccessivamente dispendiosa, con costi varianti in termini di efficienza ed accuratezza: è il caso di *script* che non necessitano di applicazioni di *testing* GUI (*Graphic User Interface*). È possibile muoversi in direzione di economizzare il prodotto di *testing*, ma questo non può rendere il prodotto economico in termini di risorse quanto un test manuale. I costi individuabili nella creazione di un proprio prodotto di test sono da ricercarsi ad esempio non solo nella manodopera, ma anche, nell'istruzione del personale e nell'allontanamento dello stesso dall'attività reale dell'impresa che si concentra per tempi generalmente lunghi, in attività differenti dalle richieste dei clienti con perdite di guadagno non indifferenti [6].

L'automazione del test, che richiede ad ogni modo uno studio preventivo fatto dall'uomo ed un controllo continuo, è utile nel caso di test di regressione, ovvero di test da ripetersi identici su più versioni di un prodotto. Tuttavia l'automazione del test è una cosa che richiede una certa quantità di tempo e che spesso fa nascere tutta una serie di problemi al momento della ripetizione del test. Le specifiche, l'interfaccia, il numero dei campi ed altri aspetti possono, infatti, cambiare

da versione a versione. Gli errori ottenuti con questo metodo, poi, sono spesso assai prevedibili e riproducibili in tempi rapidi da un test manuale. Bisogna quindi valutare attentamente la fattibilità e la convenienza prima di eseguire un test automatizzato, che si rivela efficace solo per degli utilizzi mirati e specifici [3, 8, 12].

Capitolo 2

Strategie per il testing di interfacce grafiche

I costi relativi alla verifica e validazione di sistemi software sono tipicamente sottostimati. Fattori quali l'aumento della complessità, tempi di consegna del prodotto troppo stringenti e la mancanza di ben definiti requisiti contribuiscono al fatto che l'attività di *testing* copre tra il 40% e il 70% dell'intero ciclo di vita del prodotto. Per quanto riguarda, il *testing* di sistemi software che includono interfacce grafiche, essi introducono delle ulteriori problematiche da risolvere.

Le GUI (*Graphical User Interface*) tipicamente hanno un elevato numero di potenziali ingressi e sequenze di ingressi. Per esempio, un telefono cellulare ha un gran numero di stati ed input possibili. Per cui testare un tale sistema rende necessario un numero cospicuo di casi di test. Inoltre, effettuare tutto ciò manualmente ha un costo elevatissimo e in alcuni casi risulta anche impossibile. Si rende necessario il poter eseguire il processo di *testing* in maniera automatica. Molte imprese hanno difficoltà nell'applicare il *testing* automatico su sistemi aventi interfacce grafiche, poiché, non è sempre di semplice applicazione e non vi sono attualmente *tools* efficaci che lo supportino.

Secondo *Fewseter & Graham*, gli strumenti per il *testing* non possono rimpiazzare l'intelligenza umana nel *testing*, ma senza di essi, il *testing* di sistemi complessi richiede costi troppi elevati [12]. Ci sono dei prodotti commerciali che supportano il collaudo di interfacce grafiche, molti dei quali si basano sul *Captu-*

re&Playback (vedi paragrafo 2.1.2). Purtroppo con tali tipologie di *tool* basta una piccola modifica nelle funzionalità dell'applicazione o della GUI per renderli inefficaci, inoltre, tali strumenti non aiutano ad organizzare l'attività di test, né tantomeno, offrono alcun tipo di informazione relativa alla copertura delle funzionalità da testare sull'interfaccia utente.

2.1 Progettare per la testabilità

In Figura 1 viene illustrata una rappresentazione concettuale del sistema da testare che include una interfaccia grafica. Tale tipologia di sistemi sono tipicamente composti da due o più parti: il codice per la GUI, il codice della *business logic* e il codice per l'elaborazione dei dati. Gli input generati dall'utente avvengono principalmente tramite *mouse* e tastiera. L'accoppiamento codice per l'interfaccia con quello per l'elaborazione dei dati è, spesso, causa di un significativo impedimento al *testing*.

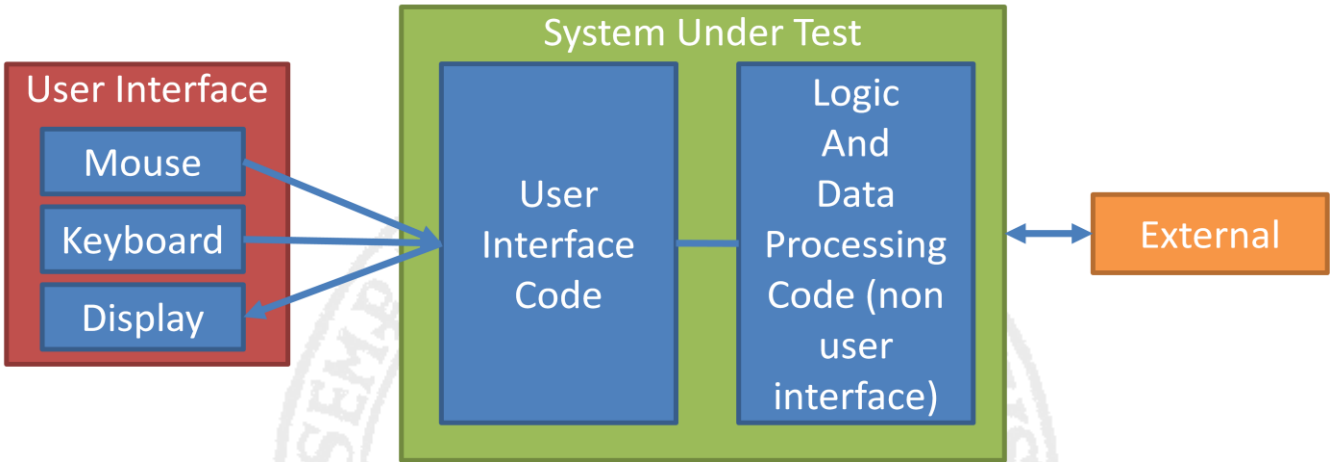

Figura 1 Elementi per il *testing* dell'*User Interface*.

Quando si collauda una GUI, sono tre le proprietà che determinano la testabilità [13]:

- **predicibilità**, misura quanto è difficile determinare il risultato atteso dopo l'esecuzione di un test;
- **controllabilità**, misura quanto è difficile fornire gli input al sistema durante la sua esecuzione;

- **osservabilità**, misura la difficoltà nel catturare e determinare la correttezza del risultato prodotto.

La complessità e la consistenza di un'applicazione determinano il grado di predicibilità. Man mano che la complessità aumenta e la consistenza diminuisce, si rende assolutamente necessario avere a disposizione una documentazione che descriva nei dettagli le caratteristiche e i comportamenti del sistema da dover comprovare. Progettare un sistema per la testabilità permette non solo di agevolare l'attività di test, ma si presta anche a supportare il *testing* automatico.

Una volta che l'architettura del sistema è stata progettata ed implementata, risulta veramente difficile modificarla per renderlo testabile. Per tale ragione, occorre, che già nelle fasi iniziali di sviluppo sia già stato tutto progettato per la testabilità dell'intero sistema da realizzare.

Un altro espediente per facilitare la testabilità e per poter aumentare la controllabilità e l'osservabilità, è quello di separare l'interfaccia grafica dalla parte logica. La maggior parte delle funzionalità dell'applicazione dovrebbero poter essere testate indipendentemente dall'*user interface*. Il motivo è che i cambiamenti apportati al *layout* della GUI non devono impattare sulla *business logic*, viceversa, modifiche della parte elaborativa non devono necessariamente protrarsi sull'interfaccia grafica.

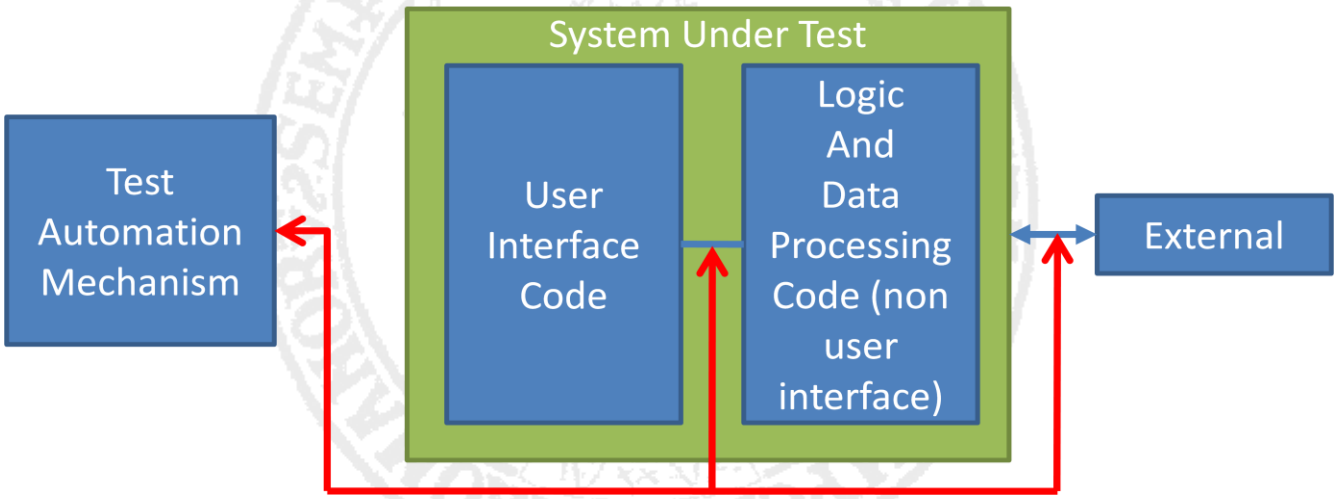

Figura 2 Interfaccia tra *GUI* e *data processing*.

In Figura 2, è rappresentata l'interfaccia tra la GUI e la parte elaborativa. Un'interfaccia ben definita apporta, in maniera positiva, un sostanziale miglioramento della controllabilità e dell'osservabilità del sistema da collaudare. I tester possono utilizzare le interfacce per inizializzare il sistema, fornire gli ingressi e memorizzare i risultati prodotti dal test. Se il test si compone di una sequenza di eventi, l'interfaccia permette di verificare la correttezza dei risultati intermedi prodotti. Questo approccio è alla base del test automatico, esso permette la netta separazione delle attività di test mirate alle interfacce utente da quelle della parte elaborativa .

Dato che è praticamente impossibile effettuare direttamente il *testing black-box*[3] di un'applicazione a livello di sistema, è possibile eseguire il test a più livelli; per cui, gli sviluppatori debbono effettuare il *testing* a livello di unità. Gli sviluppatori assieme ai *test engineers* devono, poi, eseguire il test a livello di componente ed il test d'integrazione. Infine, va effettuato il *testing* a livello di sistema e quello di accettazione. I test eseguiti sulle GUI sono a livello di sistema e permettono non solo di dimostrare le funzionalità del prodotto, ma anche, di ridurre in maniera significativa il numero di casi di test da condurre per verificare e validare le funzionalità offerte dal data *processing*. Tale approccio apporta una riduzione del *testing* manuale, impattando positivamente sui costi, necessitando, però, di una particolare architettura di sistema che lo supporti.

Test automatico

Il test di interfacce grafiche è in tutti gli effetti un test di accettazione dell'applicazione, in quanto ci si pone allo stesso punto di vista dell'utilizzatore. Tutte le funzionalità offerte dall'applicazione, possono essere invocate direttamente dalla GUI, quindi, il collaudo di interfacce grafiche permette la copertura delle funzionalità offerte dall'intera applicazione sotto test. Dato che il *testing*

[3] Il *testing black-box* è effettuato accedendo al software solamente tramite le interfacce utente, oppure tramite interfacce di comunicazione tra processi.

manuale di interfacce grafiche è tendenzialmente laborioso, c'è un gran bisogno di ridurre i costi introducendo l'automazione. I *tool* più diffusi, di questo genere, si basano sulla funzionalità del *capture/replay*. Alcuni strumenti per il *testing* memorizzano le coordinate del mouse e le azioni eseguite dall'utente come se fossero dei casi di test da riprodurre. Il problema di tali *tool* consiste nel fatto che è sufficiente una piccola modifica sull'interfaccia del *SUT* per comportare la ridefinizione dei casi di test precedentemente memorizzati. Un altro approccio sarebbe quello di catturare gli oggetti della *GUI* su cui si effettuano azioni, anziché, memorizzare le coordinate del puntatore del mouse. Sebbene la replica sia automatica, lo sforzo profuso con questi strumenti di supporto è la creazione dei casi di test e l'individuazione dei difetti. Un altro approccio, abbastanza popolare, è quello di invocare i metodi del codice sottostante come se essi fossero invocati direttamente dalla GUI.

Le attuali tecniche utilizzate dai *tool* per il *testing* di interfacce grafiche, utilizzati nella pratica, sono praticamente incompleti, e una gran parte del lavoro è demandato alle procedure manuali.

Una serie di ricerche stanno mostrando che un'alternativa, più efficace di quella del *capture&replay*, consiste nell'adoperare l'*Model-Based-Testing*[4]. Per prima cosa occorre progettare gli oggetti della GUI in maniera tale da essere sottoposti al *testing* automatico. Linee guide in merito a ciò vengono di seguito indicate [14].

- Assicurarsi che gli strumenti utilizzati per il testing sono compatibili con gli strumenti adoperati per lo sviluppo della GUI.
- Utilizzare *GUI object* che supportano il *testing* automatico.
- Evitare l'uso di controlli personalizzati, a meno che le librerie di controllo personalizzati sono sviluppati per sostenere il test. In quanto, gran parte dei

[4] MBT è un processo che si suddivide in cinque fasi principali: progettazione del SUT, generazione dei casi di test astratti, concretizzazione dei casi di test astratti, esecuzione dei test sul SUT e assegnazione dei verdetti, ed infine l'analisi dei risultati ottenuti dal test.

tool automatici supportano un insieme di controlli grafici standard forniti dall'ambiente di sviluppo. Per cui prima di prevedere l'inserimento di controlli personalizzati sarebbe opportuno verificare l'impatto con i *tools* di *testing*.

- Definire regole di *naming* degli oggetti grafici. Assicurasi che ogni oggetto sia etichettato univocamente. Molti strumenti utilizzano tali nomi per poter interagire con gli elementi dell'interfaccia.
- Aggiungere funzionalità all'applicazione per supportare il *testing*. I progettisti devono tener in considerazione fattori per agevolare il *testing*, quali: un output dettagliato, *log* degli eventi, asserzioni, monitoraggio delle risorse, punti di test, e ganci per il *fault injection*.

2.1.1 Definizione dei requisiti dell'interfaccia grafica

Il *testing* è un'attività che permette di verificare che un sistema soddisfi i requisiti funzionali. I test vengono ricavati dai requisiti ed eseguiti in ordine per verificarli, per cui, i requisiti rappresentano la base di partenza per il *testing* funzionale.

I requisiti per la parte grafica, se indicati, sono soventemente specificati mediante dei casi d'uso. Per il *testing*, i casi d'uso vengono realizzati descrivendo uno scenario in cui si evidenziano i comportamenti del sistema e diversi scenari alternativi. Questi scenari alternativi sono relativi a particolari o inusuali comportamenti. Gli scenari di test possono essere descritti come sequenze di operazioni che definiscono, per l'appunto, una serie di interazioni con gli oggetti grafici [15], dunque, per la testabilità si rende necessario osservare quanto segue.

- Comprendere i requisiti legati all'interfaccia grafica. Come già menzionato, ci sono molti modi per poter definire i requisiti, tuttavia, può capitare che i requisiti della *GUI* non siano del tutto definiti, e i tester devono determinarli interagendo con gli sviluppatori, gli esperti e i clienti.

- Assegnare la priorità ai requisiti software. Questo è importante ai fini della pianificazione, in quanto i tempi per la consegna sono, tuttavia prestabiliti. E' meglio testare le funzionalità più critiche in maniera prioritaria.
- Modellare la sequenza di interazioni per i principali scenari di utilizzo dell'applicazione. Questo può essere complicato, ma tale compito può essere semplificato costruendo una mappa navigabile della *GUI* e uno o più *state machine* che definiscono gli stati dell'interfaccia grafica e i relativi eventi che determinano le transizioni tra i vari stati.
- Gestire gli eventi inaspettati. Esso è importante perché l'utente, che inizialmente ha familiarità con l'applicazione, può eseguire una sequenza di azioni inaspettate, che possono far emergere difetti non riscontrati in fase di *testing*.

Esempio

Vediamo adesso con un esempio come tutto questo può essere realizzato [16]. I casi d'uso catturano le funzionalità offerte dal sistema, ma essi qualche volta non catturano i modi di utilizzo e i comportamenti del sistema. Prendiamo come caso di studio la realizzazione di un sistema "**Bancomat**" e illustriamo un caso d'uso, uno scenario di utilizzo e una sequenza di interazioni. In Figura 3 è rappresentato il *use-case diagram* del sistema "**Bancomat**". Esso mostra in maniera parziale e semplificata un insieme di casi d'uso comparabili con un reale sistema di Bancomat. Nell'esempio viene gestito l'*accounting* bancario come parte del sistema di "**Bancomat**" come se fosse un attore esterno.

Per supportare il *testing*, i comportamenti richiesti devono essere identificati ed espressi in termini di qualche interfaccia. Per una prospettiva di alto livello, i casi d'uso devono denotare vari comportamenti, ma questo non è sufficiente per la costruzione dei casi di test. Maggiori dettagli vengono richiesti per formalizzare le interfacce e i comportamenti richiesti.

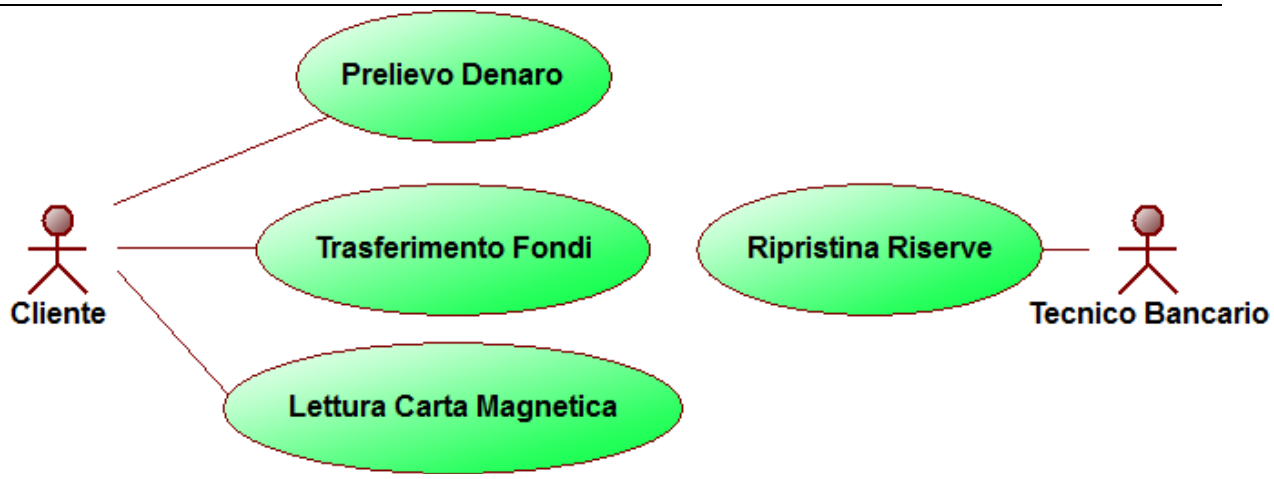

Figura 3 *Use Case Diagram* - Sistema Bancomat.

Casi d'uso

In Tabella 3 è mostrato parte di un caso d'uso per l'espletamento di un deposito di denaro. La porzione mostrata non tiene conto di tutti gli scenari alternativi.

Tabella 3 Caso d'uso di un operazione di prelievo

Nome/Azione	Prelievo Denaro
Attori	Cliente
Pre-condizione	1. La lettura della carta magnetica avviene con successo.
Post-condizione	1. Bilancio attuale = bilancio iniziale – denaro erogato – commissioni bancarie. 2. Riserva di denaro = riserva iniziale – denaro erogato.
Scenario principale	1. Il Cliente chiede di effettuare un prelievo. 2. Il Bancomat chiede al cliente quanto vuole prelevare. 3. Il cliente inserisce la quantità di denaro richiesta. 4. Il Bancomat eroga la quantità di denaro richiesta, rilascia la carta e aggiorna il bilancio attuale.
Scenari alternativi	2a. La carta del cliente è collegata a più conti correnti. 2a1. Il Bancomat chiede al cliente di selezionare uno dei conti bancari su cui effettuare le transizioni. 2a2. Il cliente seleziona il conto. 2a3. Segue dal punto 2

	2b. La carta del cliente non è collegato a nessun conto corrente affiliato. 2b1. Il Bancomat chiede se il cliente desidera spendere $2 di commissioni bancarie. 2b2. Il cliente accetta di pagare le commissioni bancarie. 2b3. Segue dal punto 2 2c. La carta magnetica letta è una carta di credito. 2c1. Il Bancomat chiede se il cliente desidera spendere $10 di commissioni bancarie. 2c2. Il cliente accetta di pagare le commissioni bancarie. 2c3. Segue dal punto 2. 4a. Il cliente chiede una quantità di denaro che non è un multiplo di $20. 4a1. Il Bancomat chiede al cliente di inserire una quantità che sia un multiplo di 20. 4a2. Segue dal punto 2. 4b. Il bancomat ha meno denaro di quanto richiesto dal cliente. (etc.)

Prioritizzazione dei requisiti

È oramai comune che i requisiti possono essere poco dettagliati o addirittura inesistenti. Si rende, altromodo oppurtuono, assegnare e definire delle priorità ai requisiti, così da far sì che lo sviluppo dei casi di test possa essere prioritizzato. Il ricorso alla prioritizzazione è utile in quanto il tempo e le risorse a disposizione sono limitati. Di seguito vengono fornite delle linee guida per poter applicare la prioritizzazione dei requisiti.

- Identificare, inizialmente, i requisiti da testare e i casi di test da sviluppare in riferimento allo scenario principale, e poi via via, quelli alternativi. Lo scenario principale definisce le funzionalità principali del sistema. Gli scenari alternativi definiscono degli eventi inusuali o inaspettati. In aggiunta, alcuni requisiti possono essere coperti da altri casi di test.

- Prioritizzare i requisiti in base alle criticità.
- Prioritizzare i requisiti tenendo in considerazione la consegna del prodotto. Se il team di testing lavora con quello di progettazione, i test possono essere prioritizzati in base all'*availability* delle funzionalità e all'importanza.

In Tabella 4 vi è un semplice esempio di prioritizzazione dei requisiti. I valori nella seconda colonna indicano il grado di priorità che diminuisce al crescere di tale valore.

Tabella 4 Esempio di prioritizzazione dei requisiti

Scenario	Priorità
Scenario principale: Prelievo di denaro	1
Scenario alternativo 2a. La carta del cliente è collegata a più conti correnti	2
Scenario alternativo 2b. La carta del cliente non è collegato a nessun conto corrente affiliato	3
Scenario alternativo 2c. La carta magnetica letta è una carta di credito	3
Scenario alternativo 4a. Il cliente chiede una quantità di denaro che non è un multiplo di $20	1
Scenario alternativo 4b. Il bancomat ha meno denaro di quanto richiesto dal cliente	1
Etc.	

La prioritizzazione dei requisiti può essere importante durante l'attività di pianificazione del *testing* e per stimare le altre fasi di un progetto. La prioritizzazione dei test da dover sviluppare e poi eseguire può essere effettuata fornendo maggiori dettagli. Tali dettagli, appunto, possono essere rappresentati da scenari, sequenze di interazione o tabelle decisionali. Ecco di seguito alcuni esempi.

- **Scenario.** Nello scenario che segue viene rappresentato il caso in cui il cliente richiede un prelievo di denaro.
    - Il cliente richiede un prelievo di denaro.

- Il bancomat chiede l'ammontare richiesto.
- Il cliente inserisce la quantità di denaro richiesta.
- Il bancomat eroga il denaro e rilascia la carta.

Questo semplice scenario mostra il comportamento del sistema durante un'operazione di prelievo effettuato da un utente. Se un test per lo scenario mostrato fosse eseguito manualmente, allora, non sarebbe necessario alcuna formalizzazione di una sequenza di interazione, altrimenti, come viene illustrato di seguito, l'analisi di una sequenza di interazione fornisce preziosi dettagli per la progettazione di casi di test.

- **Sequenza di interazione** è un utile modo di definire in dettaglio i requisiti di un'interfaccia grafica. Tale diagramma permette di evidenziare le relazioni che sussistono tra gli oggetti della *GUI* basati sugli eventi che causano la transizione tra i diversi stati che l'interfaccia grafica attraversa. Lo scenario descritto sopra viene adesso rappresentato come un *GUI map* e mostrato in Figura 4, nella quale si definiscono le diverse finestre e le relazioni tra di esse. Un *GUI map* può rappresentare vari immagini, includendo menù, *form*, e messaggi. Sebbene non sia estremamente dettagliato, la mappa è molto complessa rispetto allo scenario. La mappa mostra il come si può transitare da uno stato all'altro in base al susseguirsi degli eventi, per esempio, se l'utente inserisce il PIN non corretto, ci sarà una schermata che permetterà all'utente di re-inserirlo per un numero precisato di volte, prima che il bancomat annulla l'eventuale transizione, senza rilasciare la carta magnetica. Se l'utente seleziona una somma da prelevare maggiore del suo saldo corrente, il sistema visualizza una finestra indicante che il proprio saldo eccede e, se si vuole, selezionare una quantità inferiore. Se l'utente seleziona "Trasferimento" o "Estratto Conto" il sistema mostrerà le relative schermate. Come si può notare, ci sono almeno otto differenti sequenze di interazione che riflettono tale mappa.

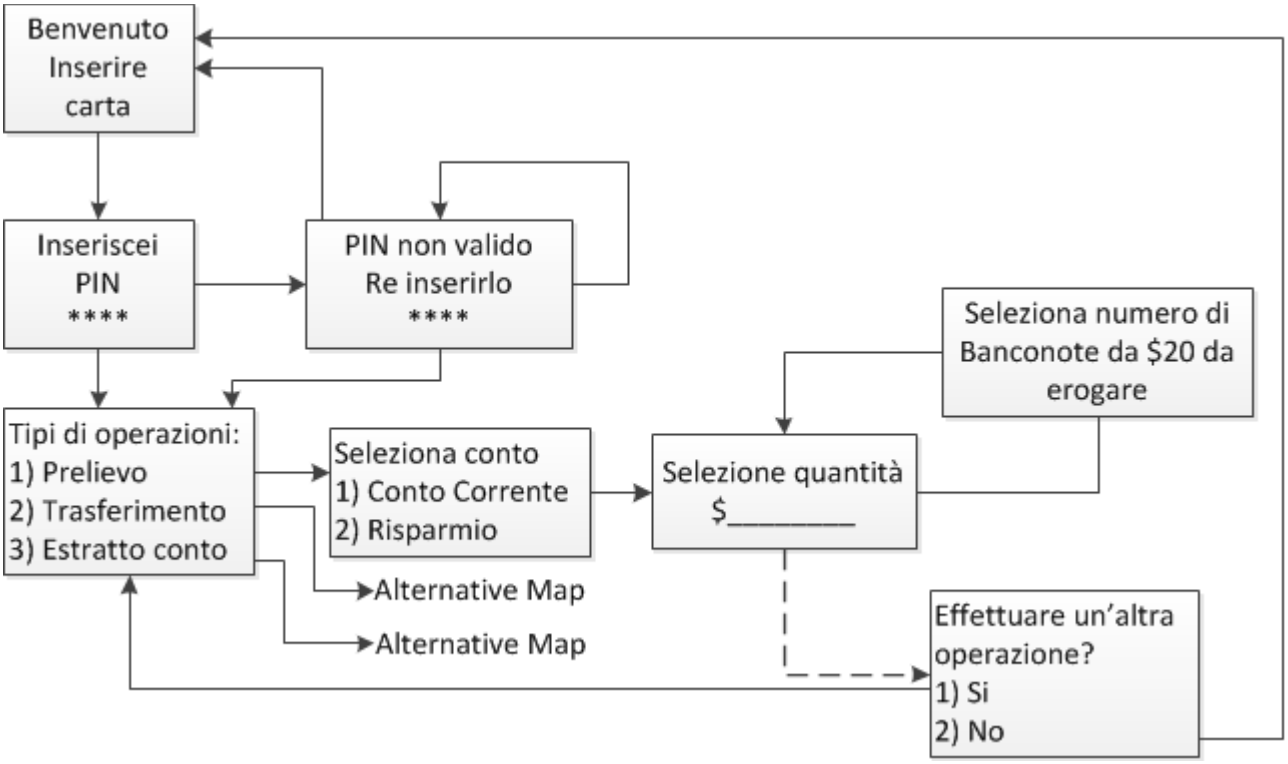

Figura 4 Mappa grafica parziale per uno scenario di prelievo di denaro.

2.1.2 Approcci al test automatico

La fase di *testing* può essere suddivisa in più attività. La seguente lista mostra in ordine di maturità le varie attività di *testing*:

- Esecuzione dei test e analisi dei risultati;
- Progettazione dei test, permette di determinare lo stato iniziale, i valori d'ingresso, e i valori attesi;
- Pianificazione dei test, permette di derivare dai requisiti del sistema cosa bisogna testare e in che ordine;
- Gestione del test;
- Misurazione del test, permette, grazie a particolari tecniche, di misurare la complessità del sistema e di valutare il grado di copertura dei test.

Esistono diversi strumenti che supportono tali attività. L'attività di progettazione dei test è assai dispendiosa e difficile da automatizzare con ben pochi *tool* di supporto.

L'efficacia e la fattibilità dell'esecuzione di test automatici dipende dalla testabi-

lità del *SUT*. Attualmente, soprattutto per quanto riguarda il test di interfacce grafiche, la progettazione e l'esecuzione di test è effettuata manualmente. Tale manualità è richiesta nella determinazione dell'*entry* del *test case* e dell'associazione dei dati. Assicurare che tutte le combinazioni logiche vengano testate richiede notevoli competenze umane e un notevole dispendio di tempo. Questo tipo di creazione di test è incline ad errori, in quanto, i tester possono incautamente ripetere gli stessi casi mentre altri non vengono per niente considerati.

Esecuzione dei test

Per l'esecuzione del *testing* funzionale esiste un modello da seguire:

- inizializzare il SUT;
- Per ogni test case:
    - Inizializzare l'output atteso (se possibile);
    - Selezionare gli ingressi;
    - Eseguire il SUT;
    - Catturare l'output e memorizzare il risultato.

Test scripting

I test *script* sono generalmente meccanismi che automatizzano il processo di esecuzione del test. Molti *tool* per l'esecuzione dei test sono basati su particolari *script* che avviano il test. I tester possono sviluppare tali *script* utilizzando vari linguaggi di programmazione, come ad esempio: VB, C, C++, Java, Perl, Tel e Python, ma possono utilizzarne anche altri creati *ad-hoc*. Il *test scripting* tipicamente richiede interfacce di programmazione per controllare il *SUT*.

Esistono dei *tool* che supportano la creazione di *test script*, comunemente detti di terza generazione, che utilizzano dei meccanismi di astrazione per ottenere *test scripting* di alto livello. Tali *tool*, purtroppo, richiedono particolari competenze nella implementazione del codice per il *testing*.

Capture & Playback

Gli strumenti a supporto del *testing* che offrono la funzionalità del *capture & playback* memorizzano, per poi eseguire autonomamente, la sequenza di azioni effettuate da un tester sull'applicazione da collaudare. Questi test sono eseguiti per verificare che le funzionalità, offerte dal sistema, sono corrette. Sebbene l'alternativa al meccanismo di *capture&replay* sono i *test script*, la stragrande maggioranza dei tester preferiscono utilizzare tali *tool* anziché mettere mano al codice. Il primo beneficio, che apporta l'utilizzo di tali strumenti, è che si prestano efficacemente ad effettuare il *testing* di regressione[5]. Ci sono diversi difetti nell'utilizzo del *capture&playback*, ed essi sono:

- Quando le funzionalità del sistema cambiano, i casi di test realizzati con la tecnica del *capture&playback* possono essere invalidati e quindi si rende necessario doverli ri-definire.
- Gli strumenti per il *capture&playback* sono progettati per memorizzare tutti gli eventi generati su particolari oggetti o in posizioni assolute. Nel primo caso vi è un minimo di prevenzione di invalidazione degli *script* quando gli oggetti in un'interfaccia grafica vengono riposizionati. Mentre, nel secondo caso, la funzione del cattura, memorizza la posizione assoluta del *pixel* per ogni evento generato. Utilizzando tale approccio basta un semplice riposizionamento di un pulsante di qualche decina di *pixel* per invalidare il test script realizzato in precedenza.
- Come già menzionato nel paragrafo 2.1, l'effettivo utilizzo dei *tool* che offrono la funzionalità del *capture&playback* dipende dall'ambiente su cui tale *tool* dovrà operare. Per cui, i tester hanno il compito di ricercare lo strumento di supporto che meglio si adatti alle proprie esigenze [17].
- La maggioranza di tale tipologia di *tool* offrono la possibilità di aggiungere

[5] Nel software capita spesso che l'introduzione di nuove funzionalità a un vecchio prodotto comprometta il suo corretto funzionamento. Pertanto, per assicurarsi che la qualità del prodotto resti immutata bisogna ripetere l'intero collaudo ad ogni modifica. Il collaudo di funzionalità preesistenti e già precedentemente testate, eseguito per assicurare che modifiche al prodotto non ne abbiano compromesso la qualità, si chiama "collaudo di regressione" (in inglese, "*regression testing*"), in quanto si vuole verificare se la qualità sia regredita.

degli *script*, grazie ai quali gli ingegneri possono modificare e manutenere come se fosse codice applicativo, purtroppo tale caratteristica richiede particolari capacità di programmazione [18].

Alla luce di tutto ciò, si può dedurre che, gli strumenti che si basano sul meccanismo del *capture&playback* offrono ben pochi benefici nell'effettuare test automatici.

Approccio *Data-Driven*

L'approccio *Data-Driven* si basa su dati memorizzati o su in *database* o su di un foglio elettronico. In questo modo vengono eseguiti con lo stesso *script* più collaudi di software ognuno dei quali utilizza differenti insiemi di dati. Un vantaggio a tale approccio è che i dati possono essere modificati senza dover per forza, modificare anche lo *script*.

Approcci basati sugli eventi generati da tastiera

Gli approcci basati su tastiera (anche detti "basati su azioni") aiutano gli ingegneri nello sviluppo e nella manutenzione dei test. Il progettista che possiede un'ottima esperienza del *SUT* e poche competenze di programmazione può, mediante l'esecuzione di talune azioni da tastiera, definire i casi di test. Ogni azione viene mappata e implementata all'interno di uno *script*. Tali tipologie di *tool* combinano i vari frammenti di *script* associati alle azioni intraprese, per poi, eseguire il caso di test. In questo modo il test automatico diventa molto semplice da realizzare.

Approcci basati su finestre

L'approccio basato su finestre deriva dall'approccio del *capture/playback*, dove, in questo caso, gli oggetti della GUI sono associati a dei parametri contenuti in un foglio elettronico. I tester definiscono lo scenario di una finestra e associano un insieme di valori per coprire i vari casi di test. L'esecuzione dello *script* trasferisce i valori, prelevati dal foglio elettronico, sui vari oggetti della GUI dell'applicazione da collaudare.

Approcci orientati agli oggetti

Gli approcci orientati agli oggetti, introducono all'interno degli *script* di collaudo, concetti derivati dalle tecniche di programmazione *Object Oriented* (OO) e aggiungono funzionalità che permettono di controllare la GUI. Possono essere definiti dei nuovi oggetti e classi, in base a funzionalità personalizzate. Tale approccio è equivalente ad un approccio basato sugli eventi generati da tastiera dove, in questo caso, il codice dei casi di test sono associati ad attributi e metodi degli oggetti da controllare.

Monkey testing

Le applicazioni che offrono un'interfaccia grafica offrono meccanismi che permettono di gestire eventuali *input* non corretti, tali meccanismi guidano l'utente nella giusta direzione. In tali applicazioni, esistono delle particolari sequenze di eventi che portano il sistema in uno stato non valido, determinando così, il *crash*. Qualora l'organizzazione, che deve eseguire il collaudo, non ha abbastanza competenze nel trovare gli eventuali *bug* del sistema, potrà utilizzare il *Monkey Testing*. Tale approccio consiste nel generare dei casi di test, che a loro volta, generano eventi casuali sulle interfacce grafiche come ad esempio click, inserimento di valori, etc., al fine di portare il sistema ad un eventuale *crash*. Durante l'esecuzione di tale test viene memorizzato il log delle azioni compiute al fine di individuare la sequenza di eventi che hanno causato l'anomalia. *James Tierney*, responsabile dell'attività di *testing* di Microsoft, in una sua presentazione ha comunicato che la società in cui opera utilizzando il *Monkey Testing* sono riusciti ad individuare dal 10 al 20% dei *bug* totali. Tale approccio, purtroppo, è assai limitato: permette solo di trovare quegli errori che causano un *crash* del sistema, non permette di verificare la sua correttezza [19].

Model-Based Test Generation

Esistono diversi approcci basati su modelli di test per supportare il collaudo di applicazioni dotate di interfacce grafiche. Molti di questi approcci automatica-

mente estraggono sequenza di test analizzando modelli di macchine a stati finiti. Una volta che il test è stato generato, viene trasformato da tali strumenti in un *test script*, che esegue il collaudo in maniera molto similare ai *tool* di *capture/playback*. Il punto di forza di queste tecniche sta nel fatto che, la generazione dei test può guidare tutte le possibili sequenze associate ad un *FSM* che rappresenta l'applicazione da collaudare [20].

2.1.3 Altre considerazioni

Alle problematiche su esposte, se ne aggiungono altre, che subentrano nel momento in cui si opta per il *testing* automatico.

Terminazione dell'attività di testing

L'attività di *testing*, di solito, termina quando sopraggiunge la data di consegna del software da rilasciare. Questo comporta che il software realizzato non sia di qualità elevata, andando a impattare anche sul produttore. Ci sono, altresì, diverse soluzioni per poter decidere quando l'attività di *testing* può essere considerata terminata.

Generalmente, poichè, la pianificazione dell'attività di collaudo si basa sui requisiti, il miglior modo di determinare la terminazione è quando tutti i requisiti sono stati testati. Questo implica che occorre verificare effettivamente che i requisiti vengano coperti. Tale attività di verifica avviene nella pianificazione del *testing*, per poi, successivamente, andare a misurare le sequenze d'interazione coperte dai casi di test.

Un'altra soluzione consiste nell'utilizzare un approccio strutturale della copertura dell'attività di collaudo. Tale approccio misura che una precisa percentuale del sistema sia stata interessata dall'esecuzione dei casi di test. Questa percentuale è ottenuta tracciando i percorsi eseguiti dai *test case* in relazione con il totale dei percorsi ammissibili.

Occorre sviluppare, dunque, i test in maniera da coprire tutte le possibili sequen-

ze d'interazione, in base al grado di priorità che essi hanno. Se il tempo e le risorse lo permettessero, sarebbe opportuno effettuare quanto più possibile il *fault-based test* per individuare delle sequenze di interazione utente illegali. I progettisti dovranno anche identificare le sequenze d'interazione che possono essere verificate indipendentemente. Questi test possono essere rappresentati da un macro stato, riducendo, così, il numero di percorsi possibili nell'FSM dell'applicazione. Quando il sistema subisce una modifica, bisogna verificare gli impatti che si hanno sulle funzionalità del sistema ed effettuare il test di regressione utilizzando ciò che già è stato sviluppato. Il test di regressione serve ad assicurare che le modifiche apportate al software, non abbiano introdotto dei difetti a parti collaudate già in precedenza. Spesso, accade che il test di regressione viene eseguito non completamente, e ciò comporta che se i difetti vengono riscontrati durante l'esecuzione del nuovo sistema, i costi per la loro risoluzione, aumenteranno in maniera considerevole.

Quando si utilizza un approccio al *testing model-based*, poichè il modello può essere aggiornato molto semplicemente, i test possono essere rigenerati per poi essere ri-eseguiti automaticamente. Questo si traduce in una riduzione dei costi relativi al *testing* di regressione.

Modifiche nei processi e nell'organizzazione

In molte organizzazioni si sta iniziando ad ingegnerizzare l'intera attività di *testing*. Tale attività, inizia, durante la fase di analisi dei requisiti, in cui si comprende al meglio la testabilità del prodotto. Quando i tester interagiscono con i committenti, come ad esempio gli utenti, i progettisti possono apportare delle migliorie individuando requisiti ambigui, o non documentati. Nell'interpretazione dei requisiti, i tester possono individuare quelli non testabili. Lavorando in sinergia con il processo di sviluppo, i tester possono aiutare gli *stakeholders* a rifinire i requisiti, assicurando che siano testabili. In aggiunta, le interazioni che avvengono con i progettisti possono dare conferma che il sistema

sia stato progettato per la testabilità e che supporti il test automatico.

2.2 Comparazione dei Tool per il Testing Automatico

Come già accennato, la prima parte del mio lavoro di tesi si è concentrata sulla ricerca di strumenti, che offrono supporto al *testing* automatico di applicazioni dotate di interfaccia grafica scritte in Java. La ricerca si è focalizzata principalmente su applicazioni open source, in quanto si prestano ad essere modificate e adattate alle specifiche esigenze. I *tool* che più si avvicinano alle nostre esigenze sono stati i seguenti [21]:

- **Cacique** è un *tool* che permette l'automatizzazione del *testing* senza chiedere particolari conoscenze di linguaggi di programmazione da parte dell'utilizzatore. Esso permette di sviluppare e memorizzare, in appositi *repository*, gli *script* realizzati. È possibile, inoltre, eseguire i test su diverse piattaforme e con diversi ingressi prelevati da apposite strutture dati. Infine, è stato concepito per poter concatenare più *script*. **Cacique** si basa su Selenium[6] per effettuare il *recording* dei *test script* [22].
- **Concordion** offre supporto per la realizzazione di test di accettazione. I test *script* vengono codificati in linguaggio Java ed integrati direttamente con JUnit[7]. **Concordion** crea automaticamente la documentazione del test, e ad ogni esecuzione degli *script* viene aggiornata [23].
- **Cucumber** è uno strumento che a partire dalla descrizione dei test funzionali è in grado di generare i relativi *script*. Il linguaggio che **Cucumber** utilizza è chiamato **Gherkin**. Esso può essere utilizzato per testare applicazioni indipendentemente dal linguaggio di programmazione utilizzato. L'unico requisito è che occorre avere delle basilari conoscenze del linguaggio Ruby[8] [24].

[6] *Selenium* è una suite di strumenti specifici per effettuare il test della applicazioni web. Con particolari estensioni è possibile creare ed eseguire in maniera semplice e veloce dei test, che possono poi essere esportati in molti linguaggi di programmazione.
[7] *JUnit* è un *framework* creato da Kent Bech e Erich Gamma per il test di unità di applicazioni scritte in Java.
[8] *Ruby* è un linguaggio di programmazione *open-source* dinamico che da particolare rilevanza alla semplicità e alla produttività, dotato di una sintassi elegante, naturale da leggere e facile da scrivere.

- **GraphWalker** (ex org.tigris.mbt) permette di generare sequenze di casi di test partendo da un FSM relativo all'applicazione da collaudare. È dotato di due modalità per la generazione delle sequenze di test, una *offline* e una *online*. Una delle sue peculiarità è che si utilizza un formalismo proprietario, diverso ma più semplice rispetto ad UML, per la definizione di FSM. L'idea alla base di **GraphWalker** è che possono essere coperti tutti i percorsi individuabili dall'FSM. Utilizzando la modalità *online* è possibile generare la sequenza di test andando a determinare a *runtime* quale cammino intraprendere durante l'esecuzione del test vero e proprio. Esso, inoltre, è *event-driven*, ossia offre la possibilità di cambiare modello in base all'evento verificatosi, per esempio, se abbiamo un modello relativo alla navigazione dell'interfaccia grafica di un telefono cellulare, se per caso sopraggiunge una chiamata allora si passa, ad esempio, al modello che gestisce la chiamata [25].
- **Harness** è un *framework Objet Oriented* che facilita la scrittura di test funzionali in Java. Esso si basa sul concetto di casi di test e utilizza XML per individuare il formato degli *input* e degli *output* [26].
- **IdMUnit** è un *framework* per il *testing* di unità automatico. Permette di eseguire il test di regressione, definendo i passi e il formato dei dati mediante fogli elettronici [27].
- **iValidator** è un *framework* scritto in Java atto a supportare il *testing* di regressione di applicazioni non necessariamente Java. I *test suite* vengono descritti mediante creazione di documenti XML, con cui è possibile definire il flusso e la struttura dei dati. Inoltre, **iValidator** permette non solo di gestire scenari complessi ma può essere utilizzato anche per realizzare il *testing* d'integrazione[9] [28].
- **Jameleon** è un *framework* di test automatizzati che possono essere facilmente utilizzati anche da utenti non esperti. Uno dei concetti alla base di **Jame-**

[9] Il test d'integrazione rappresenta l'estensione logica del test di unità. Esso consente di individuare i problemi che si verificano quando due unità si combinano.

leon è quello di creare un gruppo di parole chiave o *tag* che rappresentano differenti schermate di un'applicazione. Tutta la logica necessaria per automatizzare ogni schermata può essere definita in Java ed associata a parole chiave, che possono poi, essere raggruppate in diversi insiemi di dati, per formare gli *script* di test senza necessitare di una conoscenza approfondita sul funzionamento dell'applicazione da collaudare. Gli *script* di test vengono, poi utilizzati per automatizzare i test e per generare la relativa documentazione. **Jameleon** è stato progettato per verificare molti tipi di applicazioni. Per rendere ciò possibile, **Jameleon** è stato concepito come plug-in e progettato per dare supporto a diverse tipologie di *testing* quali: integrazione, regressione, funzionali e di accettazione [29].

- **jDiffChases** permette la comparazione di versioni diverse delle interfacce grafiche di applicazioni, evidenziando le loro differenze. Mediante la funzione del *record&play*, è possibile eseguire sulle versioni, sottoposte a verifica, gli scenari di utilizzo. **jDiffChases** compara le schermate, mostrando le differenze e documentando tutto in un *report* con le annesse immagini delle differenze [30].
- **Jemmy** è in realtà una libreria Java che permette di creare test automatici per le applicazioni Java dotate di interfaccia grafica scritta utilizzando componenti Swing[10] o AWT[11]. Gli *script* creati mediante **Jemmy** permettono di invocare azioni sugli oggetti grafici. **Jemmy** non offre funzionalità di *capture&playback*, non è dotato di interfacce grafiche, ma è dedicato al test di stabilità riuscendo a collaudare complicate e dinamiche GUI di applicazioni Java [31].
- **Marathon** è uno strumento professionale per la creazione di *suite test* automatizzate per applicazioni dotate di interfaccia grafica Java/Swing. **Mara-**

[10] Swing è un *framework* per Java che offre la possibilità di sviluppare le interfacce grafiche.
[11] AWT (Abstract Window Toolkit) è la libreria Java che contiene le classi e le interfacce fondamentali per la creazione di elementi grafici.

thon supporta i test automatici, test semi automatici e anche una modalità di test esplorativi per test manuali. La creazione degli *script* richiede, anche se in misura minore, la scrittura del codice. **Marathon** mette a disposizione le funzionalità di *recording & playback*, inoltre è possibile scrivere i *test script* utilizzando il linguaggio Ruby [32].

- **Maveryx** è uno strumento per l'automazione di test funzionali e di regressione di applicazioni dotate di interfaccia grafica scritte in Java. Esso elimina la dipendenza dalla *GUI Map*[12], anche detta mappa grafica. Gli elementi grafici vengono identificati direttamente a *runtime* durante l'esecuzione dei test script. **Maveryx** permette di identificare gli oggetti grafici anche in caso di informazioni approssimative degli elementi su cui effettuare le azioni. Inoltre lavora con Eclipse, NetBeans, IBM Rational Fuctional Tester, etc.. [33].

I *tool* sopra descritti, dopo un'attenta analisi, sono stati messi a confronto per individuare quello che più si avvicinasse alle nostre esigenze. In tabella 5 viene mostrata la comparazione dei *tool* presi in esame. Dopo un'accurata analisi dei *tool* per il *testing* automatico, si è giunta alla conclusione che **Maveryx** è la soluzione migliore.

[12] Con il *GUI map* viene definita un'astrazione dell'interfaccia grafica da testare. In essa vengono definiti i nomi e le descrizioni fisiche degli oggetti dell'interfaccia grafica. Di solito il *GUI map* fornisce un archivio di elementi grafici, con cui se il tester apporta alcune modifiche a tali elementi, automaticamente vengono modificati gli *script* che da esso sono stati definiti.

Tabella 5 Comparazione tool per il testing automatico

Applicazione	Caratteristiche comparate						
	Capture & Playback	Individuazione al runtime degli elementi grafici	Data Driven	Linguaggio utilizzato per la scrittura dei casi di test	Supporto al test di regressione	Plug-in Eclipse	Creazione automatica degli script
Cacique	✗	✗	✓	Selenium	✗	✗	✓
Concordion	✗	✗	✗	Java	✗	✗	✓
Cucumber	✗	✗	✗	Gherkin	✗	✗	✓
GraphWalker	✗	✗	✗	Java	✗	✗	✗
Harness	✗	✗	✗	Java	✗	✗	✓
IdMUnit	✗	✗	✗	n.d.	✓	✓	✓
iValidator	✗	✗	✗	Java/XML	✗	✓	✓
Jameleon	✗	✗	✓	Java	✗	✗	✓
jDiffChaser	✓	✗	✗	Java	✓	✗	✓
Jemmy	✗	✗	✗	Java	✗	✗	✓
Marathon	✓	✗	✗	Java	✓	✓	✓
Maveryx	✗	✓	✓	Java/XML	✓	✓	✓

2.3 Maveryx

Maveryx è un *tool open source* per il test automatico di tipo funzionale e di regressione per applicazioni scritte in Java. Esso fornisce al tester un insieme di librerie che permettono di eseguire azioni su elementi grafici del sistema da collaudare.

Maveryx elimina completamente la dipendenza dal *GUI map*. L'interfaccia utente viene mappata direttamente al *runtime* e gli elementi da testare vengono identificati mediante l'*Intelligent GUI Objects Finder* che utilizza alcuni algoritmi di ricerca per fornire tale funzionalità. **Maveryx** utilizza algoritmi di localizzazione di tipo *fuzzy* in caso che le informazioni fornite nei *test script* siano parziali o nel caso in cui l'elemento da testare sia cambiato. **Maveryx**, quindi, permette di ridurre significativamente i tempi e i costi nel creare e manutenere i *test script*.

2.3.1 Eliminazione dei *GUI maps*

Uno delle grandi sfide nel test automatico è quello di dover interagire con le interfacce grafiche. Molti *tool* per il test automatico utilizzano il "*GUI maps*" per riconoscere e localizzare gli oggetti da verificare. Una "*GUI map*" è una vista

statica che descrive l'applicazione sotto test. Gli strumenti che ricorrono ad essa, leggono le descrizioni fornite dalla mappa per trovare gli elementi, collocati nell'interfaccia grafica dell'applicazione da testare, che abbiano le caratteristiche descritte.

Generalmente gli oggetti da mappare per un'applicazione possono essere creati in due modi. Il primo consiste nell'effettuare una registrazione dei casi di test, il secondo nell'utilizzare dei *tool* che riescono a prelevare informazioni durante la scrittura del codice. In questo modo le mappe vengono create, costruiti e eseguiti i *test script*. Adottando tale approccio, si rende impossibile avviare il test automatico prima che l'applicazione da testare sia stata già implementata. Quando, l'interfaccia utente viene modificata si rende necessario aggiornare i "*GUI maps*" affinché riflettano tali modifiche.

Diversamente da altri strumenti, **Maveryx** analizza ed individua gli oggetti da testare e le loro caratteristiche, valutando tutto al *runtime* durante l'esecuzione dello *script*, senza utilizzare alcuna mappa grafica. Quando un test viene eseguito, così come farebbe un fotografo, **Maveryx** effettua degli "*snapshots*" dell'interfaccia utente dell'applicazione da collaudare. Ogni "*snapshot*"viene elaborato dall'*Intelligent GUI Objects Finder*. Quest'ultimo, mediante l'utilizzo di particolari algoritmi di ricerca, riesce ad identificare e a localizzare gli elementi dell'interfaccia grafica su cui vanno effettuati le operazioni richieste. **Maveryx** analizza ed identifica un oggetto nello stesso modo con cui una persona, guardando un'immagine individua un punto ben preciso nel quale è collocato l'elemento di interesse, in questo modo si possono sviluppare automaticamente i test script prima che l'applicazione venga rilasciata e pronta per il *testing*.

Un modo semplice per adattare, al meglio i *test script* qualora l'applicazione da collaudare venisse modificata, è quello di utilizzare un approccio *data-driven*. In tale modo si crea un ulteriore livello di astrazione dell'interfaccia grafica, così da separare la sequenza d'interazione, con i dati da passare all'interfaccia utente.

Tutto ciò senza dover apportare alcuna modifica agli *script*.

2.3.2 *GUI Objects Finder*

Gli elementi definiti nello script vengono identificati e localizzati direttamente al *runtime* mediante il *GUI Objects Finder* con delle particolari regole di *matching*. Questa tecnologia permette di ridurre significativamente i tempi richiesti ai tester per dover manutenere e adattare i casi di test man mano che vengono apportate modifiche al software da collaudare. Avere a disposizione uno strumento che permette di automatizzare le verifiche, offre diversi vantaggi che può condurre a dei risultati davvero soddisfacenti. Lo scrivere i casi di test in parallelo con le attività di progettazione e sviluppo non sempre è affidabile, perché i requisiti o i documenti di progettazione possono mutare, in quanto possono essere incompleti o interpretati male. L'accertarsi, in fase di progettazione o di sviluppo, che alcune informazioni siano incomplete può, inevitabilmente, impattare con le attività di *testing* rendendole, nel peggiore dei casi, totalmente inutili.

Di solito, la modifica, anche minimale, dell'interfaccia utente, come ad esempio l'aggiunta di qualche elemento, può causare la necessità di rifare da capo e/o modificare i casi di test impattati dalla modifica. **Maveryx**, utilizzando degli avanzati algoritmi di ricerca, è in grado di localizzare gli elementi grafici durante l'esecuzione di una procedura di test, nonostante che, in un secondo momento, tali oggetti sono stati modificati. **Maveryx** cerca al *runtime* gli oggetti che soddisfano esattamente o parzialmente le informazioni presenti negli *script*. Gli algoritmi di *matching* attualmente utilizzati sono elencati e descritti approssimativamente in Tabella 6.

Tabella 6 Tipologie di algoritmi di ricerca di Maveryx

Algoritmi di ricerca	Descrizione
ExactMatch	Permette di cercare un elemento grafico da testare così come esso è stato definito nello script.
CaseInsensitiveMatch	Simile all'ExactMatch ma è case insensitive.
PartialMatch	Permette di localizzare un oggetto anche se le informazioni fornite sono incomplete.
Wildcard	Permette di localizzare un elemento da testare utilizzando le espressioni regolari di Java
LevenshteinDistance	Permette di cercare un oggetto utilizzando la sua distanza di Levenshtein[13].
SynonymousMatch	Permette di cercare gli elementi da controllare, cercando all'interno di una sorte di dizionario, che abbiano delle descrizioni similari o dei sinonimi.
GoogleTranslation	Permette di localizzare gli oggetti traducendo la descrizione fornita.

Modificando l'*object-matching sensivity* è possibile assegnare dei pesi ai risultati prodotti dai vari algoritmi di ricerca, e questo senza dover mettere mano al codice degli *script*. Questa caratteristica, permette, non solo di mantenere gli *script*, ma, permette anche al tester di verificare le funzionalità dell'applicazione prima che l'interfaccia grafica sia stata completata.

Infine il *GUI Object Finder* può essere esteso con dei nuovi algoritmi di *matching* al fine di testare al meglio l'applicazione desiderata.

[13] La distanza di *Levenshtein* misura le differenze fra due stringhe. Introdotta dallo scienziato russo *Vladimir Levenshtein* nel 1965, serve a determinare quanto due stringhe siano simili. Viene applicata per ricercare similarità tra immagini, suoni, test, etc.

Capitolo 3

Progettazione del tool *Testeryx*

SELEX SISTEMI INTEGRATI è la società di **FINMECCANICA** che progetta, realizza e commercializza Grandi Sistemi per l'*Homeland Protection*, sistemi e radar per la difesa aerea, la gestione del campo di battaglia, la difesa navale, la gestione del traffico aereo ed aeroportuale, la sorveglianza costiera marittima. SELEX S.I. è fortemente impegnata anche nell'ottimizzazione dei processi interni. Dati gli innumerevoli progetti sui cui SELEX S.I. è impegnata e la grande mole di applicazioni software da dover sviluppare e testare, diventa di estrema importanza dotare l'azienda di uno strumento che supporti al meglio l'attività di *testing*. Tale supporto è visto sia nell'ottica dell'automatizzazione delle attività di Verifica e Validazione, sia nel poter generare la documentazione relativa, che deve essere stilata secondo precisi standard di formattazione. In modo tale, da utilizzarla sia per correggere gli eventuali errori individuati, sia per consegnarla al cliente e fargli costatare con quale qualità è stato realizzato il software.

3.1 Requisiti Software

Con lo scopo di mostrare soltanto i parametri più importanti che hanno caratterizzato la progettazione e lo sviluppo del *tool*, in questo paragrafo, saranno messe in risalto tutte le caratteristiche importanti che hanno portato all'attuale implementazione di *Testeryx*.

La progettazione del *tool* si basa sul rispetto dei seguenti requisiti:

- semplicità di utilizzo, garantita da un'interfaccia grafica con la quale si permette anche ad utenti non esperti di utilizzare a pieno il *tool* realizzato;
- possibilità di memorizzare i progetti e di poterli rielaborare in un secondo momento;
- generazione automatica del codice dei *test script*;
- possibilità di poter esportare la documentazione utile per l'attività di *testing* secondo opportuni standard di formattazione.

Requisiti Funzionali

- **RF1.** Il Sistema permette la creazione di "*Project Work*" in cui memorizzare e gestire tutti i dati e le informazioni inerenti alle attività di *testing*. Tali dati e informazioni si riferiscono sia alla documentazione necessaria quali SRS (*Software Requiremenst Specification*), STP (*Software Test Plan*), STD (*Software Testing Description*) e STR (*Software Test Report*), sia agli *script* necessari alla verifica e validazione del software da collaudare.
- **RF2.** Il Sistema permette l'aggregazione dei casi di test secondo criteri scelti dall'utente, in modo da gestire l'organizzazione interna del "*Project Work*".
- **RF3.** Il Sistema permette di definire la procedura di test. Essa è composta da un insieme di "*step*", da dover poi eseguire in maniera sequenziale. Ciascuno "*step*" ha:
  - un identificativo numerico che ne indica la progressività;
  - una sequenza di azioni da eseguire sull'interfaccia grafica del SUT;
  - le condizioni da verificare;
  - e eventuali annotazioni.
- **RF4.** Il Sistema permette il salvataggio delle procedure di test. In modo da poterle definire ancor prima, che il team di progettazione e di implementazione abbia prodotto qualcosa da sottoporre al collaudo.
- **RF5.** Il Sistema permette di eseguire le procedure di test, così come sono sta-

te definite, e di verificare all'istante la correttezza del SUT.

- **RF6.** Il Sistema, ad ogni esecuzione di una procedura di test, memorizza il *report* relativo. Nel *report* viene indicato l'esito della procedura di test. Nel caso in cui sono stati riscontrati delle anomalie, esse vengono riportate nel *report*. L'utente può decidere se visualizzare o meno, la reportistica mediante una finestra visualizzata al termine dell'esecuzione della procedura di test.
- **RF7.** Il Sistema permette di avviare l'esecuzione selettiva delle procedure già definite. Ovviamente, di tali procedure occorre che sia stata rilasciata la parte dell'applicazione da testare a cui esse si riferiscono. Tale funzionalità permette all'utente di effettuare il *testing* di regressione e di visualizzare il *report* al termine di tale attività. Nel caso di test di regressione, qualora una procedura fosse stata già eseguita, viene fatto il confronto con i report passati.
- **RF8.** Il Sistema permette la generazione automatica dei documenti a corredo dell'attività di test, quali STD e STR.

Requisiti non funzionali

- L'utente deve fruire di un'interfaccia grafica intuitiva.
- Il Sistema permette anche ad un utente non esperto di poter definire le procedure di test in maniera *user friendly*.
- I documenti prodotti dal Sistema vengono redatti in maniera conforme agli standard aziendali adottati, quale lo standard MIL-STD-498[14].

[14] MIL-STD-498 è uno standard militare, adoperato negli USA dal 1994, il cui scopo è quello di definire le regole per la descrizione dei requisiti software e per la stesura della documentazione. È stata la base di partenza per tutti gli standard ISO e IEEE.

3.2 Casi d'uso

Il diagramma mostrato in Figura 5 mostra in sintesi i casi d'uso relativi alle interazione che l'utente ha con il sistema.

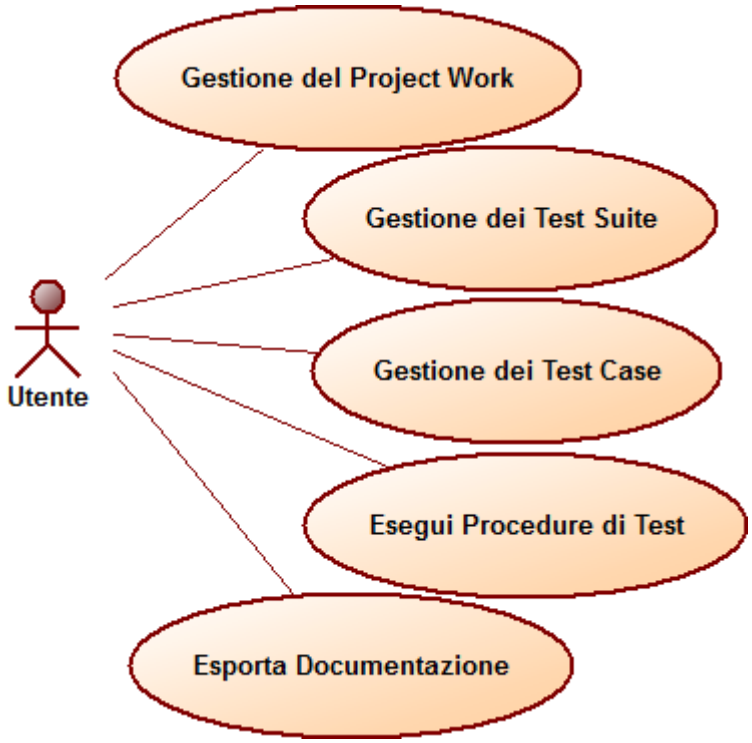

Figura 5 Diagramma generale dei casi d'uso.

Nell'analisi dei requisiti si evince la necessità di dover offrire all'utente una serie di funzionalità atte ad organizzare internamente il proprio ambiente di lavoro. È stato necessario individuare tre tipologie di gestione dell'ambiente di lavoro.

La gestione del "Project Work" fa riferimento a tutte le operazioni invocabili su di un progetto relativo all'attività di *testing*. Tale progetto può essere memorizzato o meno e le operazioni invocabili sono le seguenti:

- creazione del "*Project Work*";
- caricamento del "*Project Work*";
- modifica del "*Project Work*";
- salvataggio del "*Project Work*".

Il caso d'uso della gestione dei "*Test Suite*" è associato a tutte quelle funzionalità

che permettono:

- l'aggiunta "*Test Suite*";
- la modifica "*Test Suite*";
- il salvataggio "*Test Suite*".

Infine, il caso d'uso della gestione dei "*Test Case*" fa riferimento a tutte quelle operazioni invocabili su di un caso di test. Le operazioni da dover disporre per tale caso d'uso sono le seguenti:

- aggiunta "*Test Case*";
- modifica "*Test Case*";
- definizione "Procedura di Test";
- modifica "Procedura di Test".

Per quanto riguarda l'esecuzione delle procedure di test, tale caso d'uso, fa riferimento a tutte quelle funzionalità messe a disposizione dal *tool*. Tramite codeste funzionalità è possibile effettuare, sull'applicazione presa in esame, la sequenza di azioni e verifiche definite dall'utente all'interno delle singole procedure.

Per ultimo, vi è il caso d'uso dell'esportazione della documentazione. Le diverse funzionalità messe a disposizione dal *tool* consentono di estrapolare dal progetto definito, le relative documentazioni utili all'attività di *testing*. I documenti esportabili, quali STD e STR, verranno redatti secondo delle ben definite regole di formattazione, così da renderli facilmente consultabili.

Specializzando maggiormente la struttura dei casi d'uso generali, emergono delle nuove funzionalità. Adesso vengono mostrate una serie di diagrammi relativi alla specializzazione dei casi d'uso visti fin'ora.

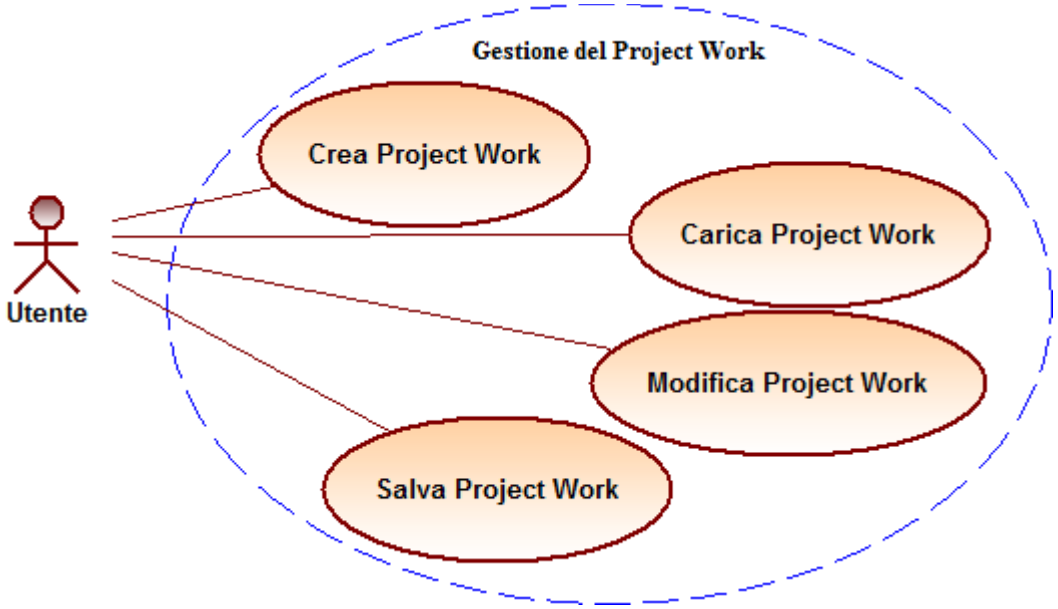

Figura 6 Diagramma di specializzazione del caso d'uso "Gestione del Project Work".

In Figura 6 è mostrata la specializzazione del caso d'uso riguardante la gestione del "Project Work". Tale caso d'uso può essere visto come generalizzazione delle seguenti funzionalità.

- **Crea Project Work**: si riferisce alla possibilità di creare un nuovo progetto in cui collocare tutti i dati e le informazioni inerenti all'attività di *testing*.
- **Carica Project Work**: si riferisce all'operazione di apertura di un progetto precedentemente memorizzato.
- **Modifica Project Work**: si riferisce alla possibilità di poter apportare modifiche al "*Project Work*". Per modifiche si intende la possibilità, da parte dell'utente, di gestire le informazioni in esso conservate.
- **Salva Project Work**: è la capacità del sistema di rendere persistente il lavoro profuso da parte dell'utente.

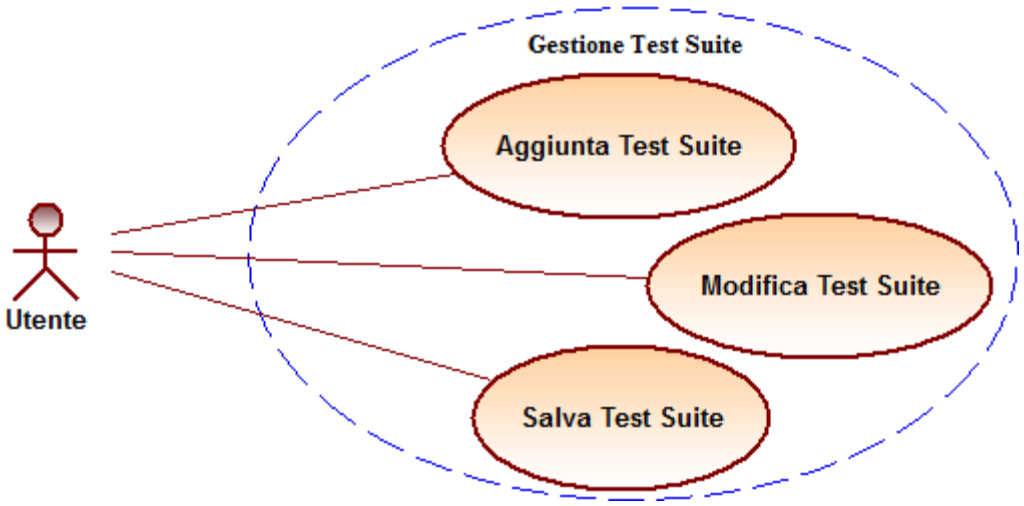

Figura 7 Diagramma di specializzazione del caso d'uso "Gestione Test Suite".

Per organizzare al meglio il progetto, i vari casi di test vengono associati a *Test Suite*. Come mostrato in Figura 7, la gestione del *Test Suite* fa riferimento a tutte le operazioni che offrono le funzionalità elencate di seguito.

- **Aggiunta Test Suite**: si riferisce all'operazione di creazione di una nuova suite di test.
- **Modifica Test Suite**: si riferisce alla possibilità di poter apportare modifiche al *Test Suite*. Per modifiche si intende la possibilità, da parte dell'utente, di gestire le informazioni in esso conservate, e di poter gestire i casi di test in esso contenuti.
- **Salva Test Suite:** permette all'utente di rendere persistente tutte le modifiche apportate al *Test Suite*.

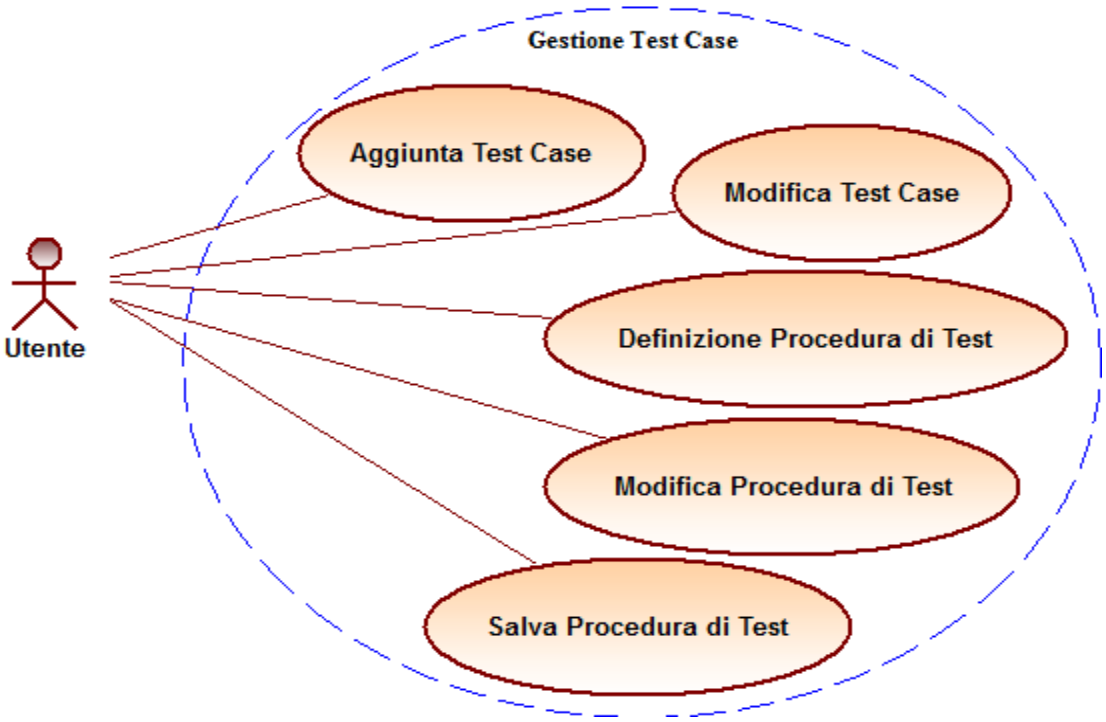

Figura 8 Diagramma di specializzazione del caso d'uso "Gestione Test Case".

Come mostrato in Figura 8, per ogni caso di test l'utente può invocare le funzionalità seguenti.

- **Aggiunta Test Case**: si riferisce alla possibilità di poter aggiungere un nuovo caso di test.
- **Modifica Test Case**: si riferisce alla possibilità di poter apportare modifiche al *Test Case*. Tali modifiche riguardano la possibilità, da parte dell'utente, di gestire le informazioni associate allo specifico caso di test e la sua riassociazione ad un altro *Test Suite*.
- **Definizione Procedura di Test**: tale funzionalità permette all'utente di definire la sequenza interattiva di azioni e verifiche da dover poi eseguire quando il SUT sarà rilasciato.
- **Modifica Procedura di Test**: si riferisce alla possibilità di modificare la procedura di test già definita.
- **Salva Procedura di Test**: permette all'utente di poter memorizzare la procedura di test. In questo modo è possibile pianificare l'intera campagna di *testing* ancor prima che il sistema da testare venga rilasciato.

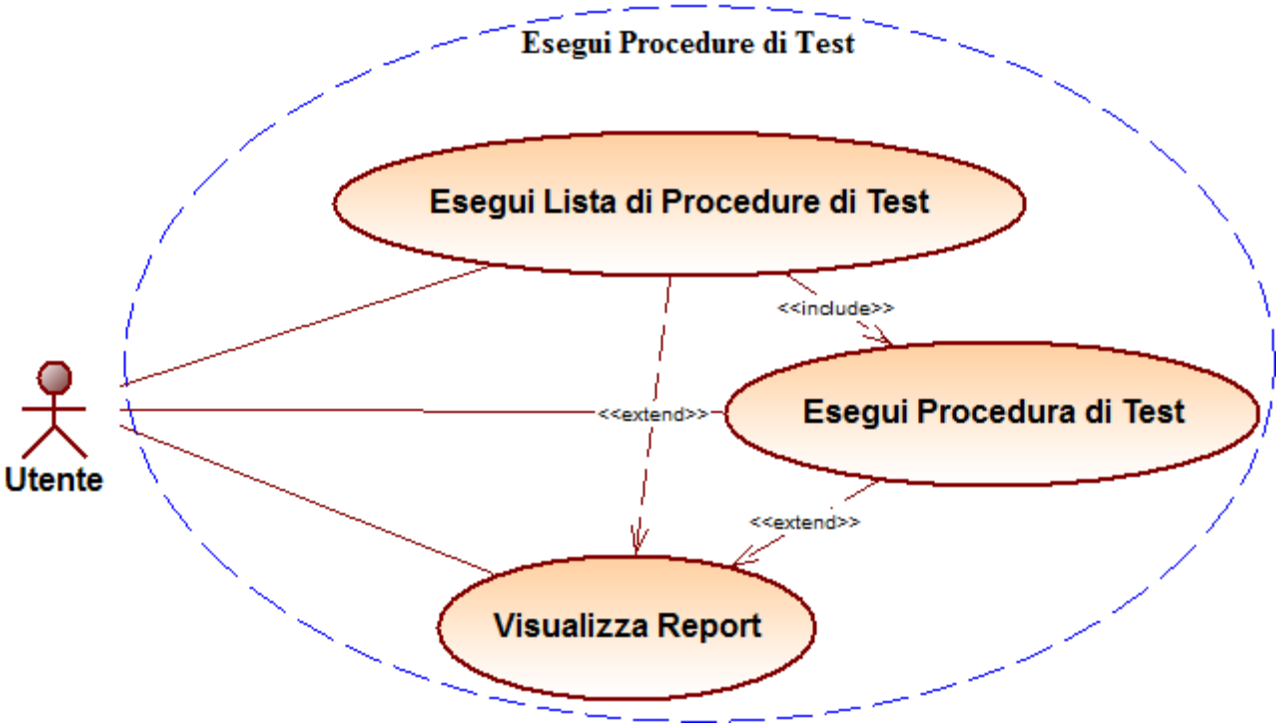

Figura 9 Diagramma di specializzazione del caso d'uso "Esegui Procedure di Test"

Il caso d'uso relativo all'esecuzione delle procedure di test permette all'utente di poter avviare l'attività di *testing* funzionale. L'utente potrà esercitare le funzionalità elencate di seguito.

- **Esegui Procedura di Test**: si riferisce all'insieme delle funzionalità che permettono al sistema di poter tradurre una sequenza interattiva, di azioni e verifiche definite dall'utente, in codice atto ad eseguire il *testing* funzionale del sistema da collaudare. Durante l'esecuzione della procedura viene prodotta la reportistica. L'utente può decidere se visualizzare o meno, al termine della suddetta procedura, la reportistica prodotta.
- **Visualizza Report**: tale funzione permette all'utente di poter prendere visione dell'esito del *testing* funzionale. Per cui, il sistema, ad ogni esecuzione di procedure di test, produrrà degli opportuni file, nei quali vi sarà indicato l'esito finale.
- **Esegui Lista di Procedure di Test**: si riferisce alla possibilità di poter avviare una campana di *testing*, selezionando quali delle procedure definite debbono essere eseguite. Tale caso d'uso può essere visto come composto dall'e-

secuzione di più procedure di test. Al termine della campagna di test, l'utente potrà prendere visione dell'intera reportistica prodotta.

3.3 Progettazione

Nel seguente paragrafo viene mostrata la progettazione del sistema da dover realizzare. Al termine del quale si avrà una completa visione architetturale e progettuale del sistema software da dover poi implementare.

3.3.1 Diagramma di Analisi

Dall'analisi dei requisiti software e dei casi d'uso si possono evidenziare le principali entità messe in gioco.

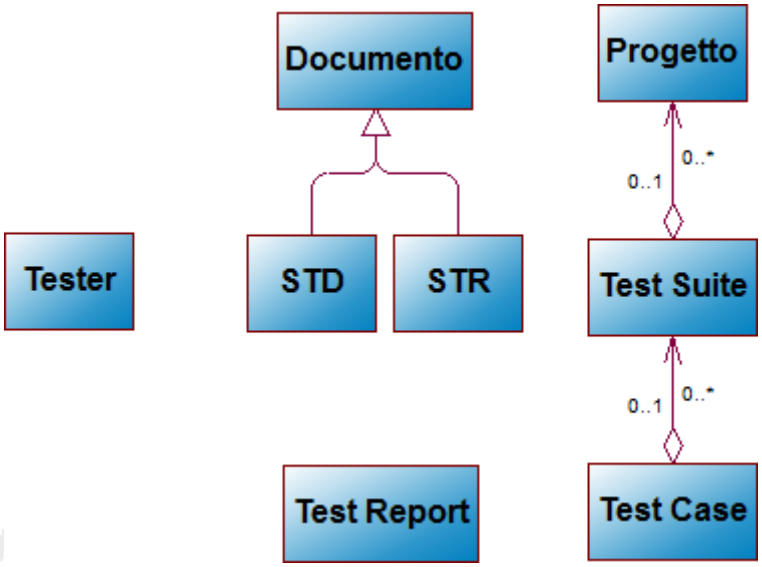

Figura 10 Diagramma di analisi del *tool Testeryx*.

In Figura 10 vi è il diagramma di analisi del Sistema da dover sviluppare. Come si può notare l'entità "Tester" permette di identificare lo specifico utente utilizzatore del *tool*. Dal momento che, si è soliti lavorare con dati altamente sensibili, tale entità è stata inserita per permettere solo ad utenti autorizzati di prendere visione di informazioni riservate.

L'entità "*Project*" offre all'utente tutte le funzionalità utili a poter gestire la propria cartella di lavoro. L'entità *Project* è composta da un insieme di *Test Suite*. A

loro volta, ogni *Test Suite* è composto da un insieme di *Test Case*. È compito di tale entità il salvataggio e il successivo *retrive* di tutti i dati forniti da parte dell'utente.

Il "*Test Report*" è l'entità cardine per poter eseguire l'attività di *testing*. Essa ha l'onere di avviare l'esecuzione delle procedure di test e di tenere traccia dei risultati monitorati.

Infine, l'ultima entità in evidenza è il "*Document*". Essa ha il compito di estrapolare dalle informazioni fornite dall'utente, tutta la documentazione da dover produrre a corredo dell'attività di *testing*. Le tipologie di documenti generati sono di due tipologie, STD (*Software Testing Description*) e STR (*Software Test Report*).

3.3.2 Diagramma di analisi

Secondo il classico modello di progettazione, tramite diagrammi UML, è previsto che le operazioni sulle classi, rappresentanti entità, vengano attivate tramite delle classi di interfaccia e vengano gestite tramite delle classi di controllo.

Una classe di tipo *boundary* ha solo il compito di contenere metodi pubblici, essi sono in grado di attivare i metodi presenti sulle opportune classi di gestione per svolgere le diverse attività, e come suggerisce il nome, hanno lo scopo di presentare un'interfaccia tra l'utente che attiva una funzionalità e la funzionalità stessa, implementata sulle classi entità corrispondenti e controllata da un opportuno gestore di tipo *control*.

Viceversa, una classe di gestione ha una struttura più complicata: i suoi metodi servono a coordinare le attività richieste tramite le classi *boundary*, richiamando, nel corretto ordine, i diversi metodi delle classi coinvolte; da un certo punto di vista un gestore può essere visto come il processo che esegue e controlla lo svolgimento di un determinato compito.

Nel problema in esame è possibile inserire quattro classi di tipo *boundary*; ciò

che però si può notare è che la scelta operata si basa sulla suddivisione in categorie di operazioni evidenziata nei precedenti diagrammi dei casi d'uso. Si utilizzano le seguenti interfacce:

- interfaccia di progetto;
- interfaccia di gestione "*Project Work*";
- interfaccia di generazione ed esecuzione dei casi di test;
- interfaccia di generazione della documentazione.

L'**interfaccia di progetto** ha lo scopo di mostrare all'utente tutte le funzionalità di gestione dei file di progetto. Tramite questa classe deve essere possibile richiamare le principali funzionalità evidenziate nel diagramma dei casi d'uso, relative ad apertura, salvataggio e creazione di un "*Project Work*".

L'**interfaccia di gestione del "Project Work"** deve fornire i metodi per l'inserimento, la cancellazione e la modifica degli elementi appartenenti alla struttura di progetto, ciò che però è richiesto dalle funzionalità di questa interfaccia è che le relative funzioni debbano adattare il loro comportamento alla categoria di nodo su cui si opera, garantendo funzionamenti diversi per nodi di tipo "*Test Project*", "*Test Suite*" e "*Test Case*". Ogni classe, associata a tali tipologie di nodi, avrà diversi parametri (che saranno schematizzati come attributi delle classi) e su di essi solo alcune funzioni potranno essere attivabili (ad esempio non è possibile associare direttamente ad un nodo di tipo "*Test Project*" un nodo di tipo "*Test Case*", ma bensì occorre aggiungere prima un nodo di "*Test Suite*" a cui associare un nodo di tipo "*Test Case*").

L'**interfaccia di generazione ed esecuzione dei casi di test** presenterà all'utente le funzionalità di generazione del codice dei *test script* ed avvierà la relativa esecuzione. Un ulteriore onere di tale interfaccia è il poter permettere all'utente di prendere visione dei dati monitorati.

Infine, sarà introdotta l'**interfaccia di generazione della documentazione**, la

quale permette all'utente di decidere quali informazioni desidera esportare e con quale formattazione redigere i documenti.

3.3.3 Scelte progettuali

Il modello architetturale utilizzato per la progettazione del *tool* è quello dell' MVC (*Model-View-Controller*).

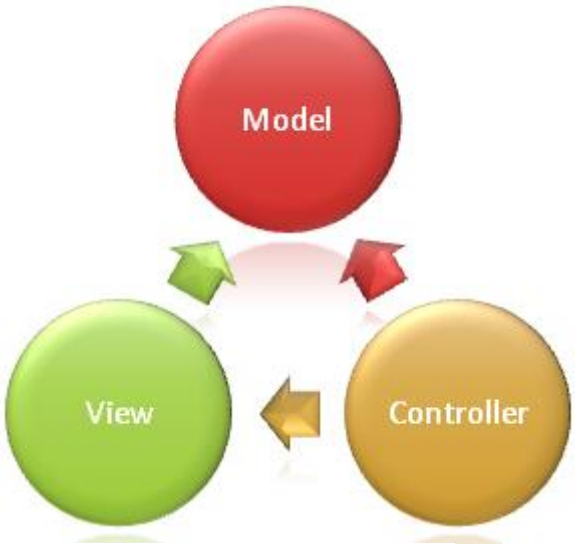

Figura 11 Vista concettuale del pattern architetturale MVC.

Il **Model-View-Controller** è un *pattern* architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software *object-oriented*. Originariamente impiegato dal linguaggio *Smaltalk*, il pattern è stato introdotto in numerose tecnologie moderne, come i *framework* basati su PHP, Ruby, Phython, Java, .NET.

Il *pattern* è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- il ***model*** elabora l'informazione ed implementa il comportamento del software;
- il ***view*** è al contatto con l'utente, recupera dati dal modello per le risposte, inizializza il controller;
- il ***controller*** accetta gli eventi traducendoli in richieste per il modello, o per le viste stesse.

Questo schema, fra l'altro, implica anche la tradizionale separazione fra la logica

applicativa (*businness logic*), a carico del *controller* e del *model*, e l'interfaccia utente a carico del *view*. I dettagli delle interazioni fra questi tre oggetti software dipendono molto dalle tecnologie usate (quali ad esempio linguaggi di programmazione, eventuali librerie, *middleware*, etc..) e dal tipo di applicazioni (per esempio se si tratta di un'applicazione *web*, o di un'applicazione *desktop*).

Tale modello architetturale risulta essere perfetto alle nostre esigenze. Grazie a tale *pattern* architetturale, le eventuali modifiche apportate su di una delle tre parti, a meno che non interessano le interfacce di comunicazione, non implicano la necessità di coinvolgere anche le altre parti.

Per quanto riguarda l'interfacciamento si è deciso di dotare l'applicazione di un'interfaccia grafica molto intuitiva. Tale scelta è totalmente in linea con il requisito non funzionale, secondo cui il sistema debba poter essere utilizzato anche da utenti non esperti.

La persistenza dei dati viene assicurata mediante la generazione di documenti XML, e nello strutturare la cartella di lavoro gestita dal *file system* del Sistema Operativo. XML è stato scelto per due motivi. Il primo è che essendo uno standard è possibile esportare i documenti prodotti e utilizzarli con altri applicativi. Il secondo motivo è che mediante l'utilizzo di *XSL*(*eXtensible Stylesheet Language*) è possibile specificare il formato di presentazione di un documento XML usando due categorie di tecniche, una opzionale che riguarda la trasformazione della struttura del documento di ingresso in un'altra struttura, e l'altra che permette la descrizione di come devono essere visualizzati gli elementi della struttura. In questo modo sarà possibile manipolare tali informazioni e quindi generare tutta la documentazione richiesta dall'utente.
Un ulteriore scelta progettuale adottata è stata quella di utilizzare per la scrittura

del codice applicativo, il linguaggio Java.

Ed infine, l'ultima scelta effettuata è relativa all'assegnazione del nome al sistema da realizzare. Dato che lo scopo principale del sistema è dare supporto al processo di *testing* e poichè si è pensato di sfruttare principalmente, le potenzialità di "*Maveryx*", uniche nel suo genere, è stato deciso di attribuirgli il nome di "**Testeryx**".

3.3.4 Diagramma architetturale

Le varie scelte progettuali, hanno portato alla suddivisione delle classi che compongono l'applicazione in più *package*, a seconda della funzione che esse offrono. Di seguito sono elencati i diversi *package* individuati.

- **Model**, in tale *package* sono state aggregate tutte le classi che permettono l'accesso ai dati utili dall'applicazione.
- **View**, appartengono a tale *package*, tutte le classi deposte a catturare gli eventi generati dall'utente e mostrare lo stato attuale del sistema.
- **Controller**, in questo *package* è stato collocato la classe che elabora gli eventi generati dall'utente, esegue alcuni controlli su di essi, ed applica le relative modifiche andando e interagendo direttamente con il *model*.
- **Utility**, infine, in tale package sono stati inserite tutte quelle classi che sono da supporto verticale a tutte classi.

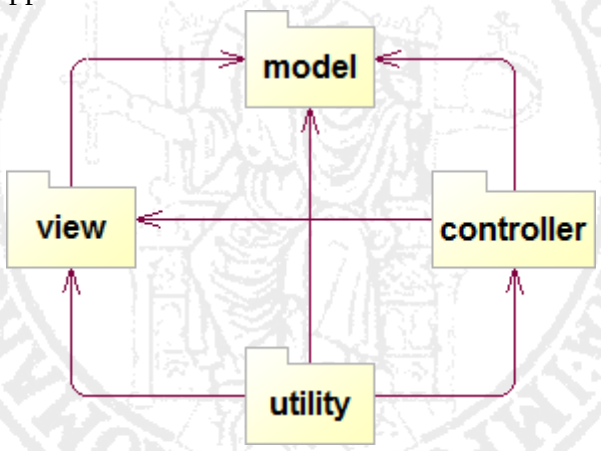

Figura 12 *Testeryx Package Diagram*.

Business logic

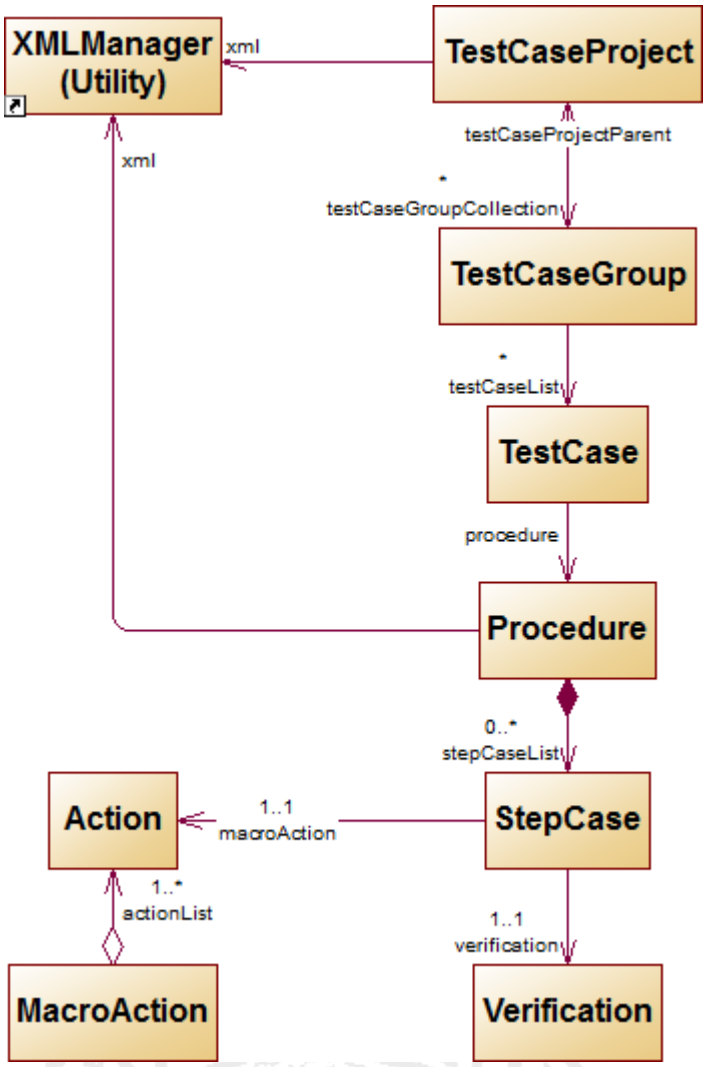

Figura 13 *UML Class Diagram* della *business logic*.

In Figura 13 vi è il *Class Diagram* relativamente al *model*. La classe principale è la "*TestCaseProject*", la quale offre le interfacce operazionali per poter accedere a tutte le altre classi. Inoltre, è importante notare che tali classi fanno ricorso alla classe "*XMLManager*" (trattata con maggiore dettaglio più avanti) per poter tenere traccia dello stato del *model*.

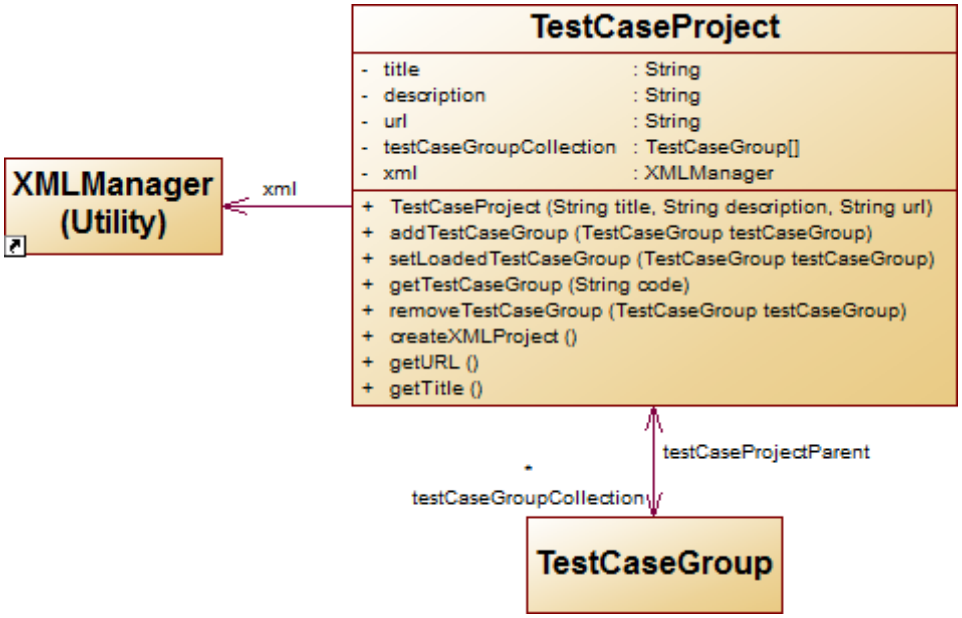

Figura 14 *TestCaseProject UML Class description*.

In Figura 14 vi è in dettaglio la struttura della classe "*TestCaseProject*". Come si può notare essa ha come parametri privati un titolo, una descrizione e un *URL*. Quest'ultimi tre parametri vengono passati dal costruttore della classe. Inoltre, tale classe mantiene ad essa associata, come riferimento, la lista dei "*TestCaseGroup*" e alla classe "*XMLManager*", con cui tiene aggiornato lo stato dell'intera *data application*. I metodi offerti da tale classe sono i seguenti:

- *addTestCaseGroup* permette l'aggiunta di una nuova *Test Suite* al progetto e poter su di esso accedere ai suoi metodi pubblici;
- *setLoadedTestCaseGroup* è un metodo utilizzato nel caricamento di un *TestCaseProject* precedentemente memorizzato;
- *removeTestCaseGrouup* permette di eliminare i *TestCaseGroup* dalla lista omonima;
- *createXMLProject* è un metodo che permette di istanziare un riferimento a se stesso nella classe "*XMLManager*";
- infine, i metodi *get* permettono l'accesso agli attributi privati della classe.

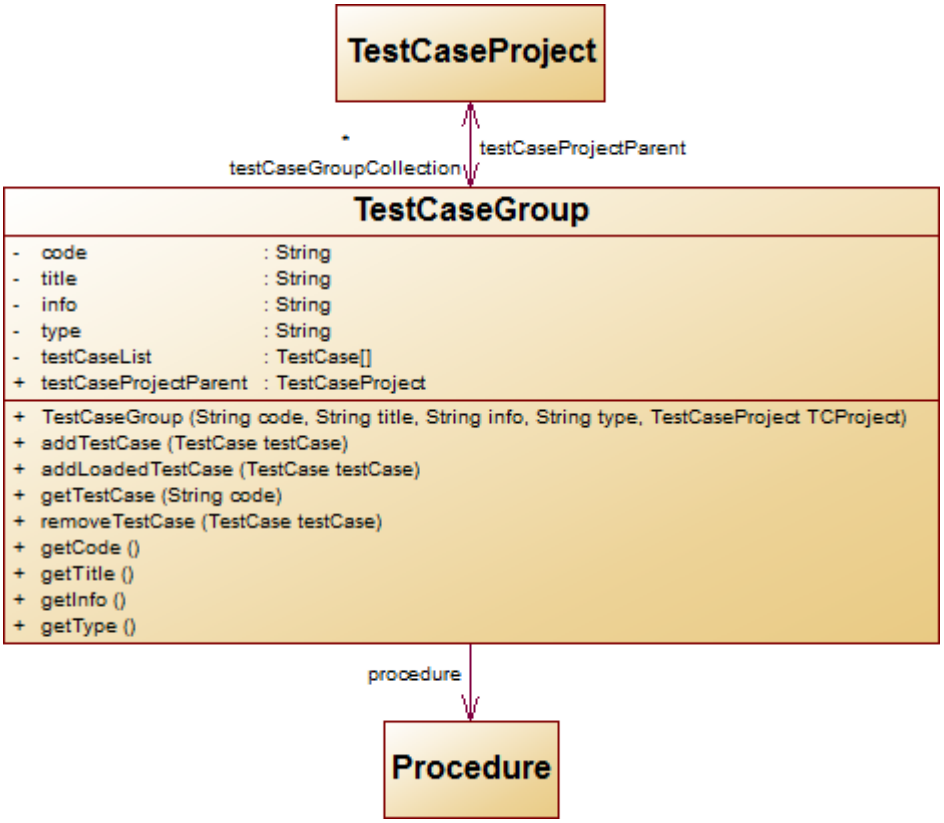

Figura 15 *TestCaseGroup UML Class description*.

La classe "*TestCaseGroup*", mostrata in Figura 15, presenta i seguenti attributi:

- *code* è un codice assegnato alla suite e tale codice viene ereditato dai casi di test associati;
- *title* è il nome mnemonico da associare alla *Test Suite*;
- *info* tale parametro permette di associare al *Test Suite* una descrizione;
- *type* permette di definire quale tipologia *di Test Case* verranno associati;
- *testCaseList* permette di tenere un riferimento a tutti i casi di test associati alla suite;
- *testCaseProjectParent* permette di tenere un riferimento al *TestCaseProject* a cui il *TestCaseGroup* è associato.

I metodi offerti da tale classe sono:

- *addTestCase* permette l'aggiunta di una nuovo *test case* al progetto e permette di invocare i suoi metodi pubblici;
- *addLoadedTestCase* è un metodo utilizzato quando un *Testeryx Project* vie-

ne caricato e occorre ripristinare lo stato dell'intero *model*;

- *removeTestCase* permette di eliminare i *TestCase* dalla lista omonima;
- infine, i metodi *get* permettono l'accesso agli attributi privati della classe.

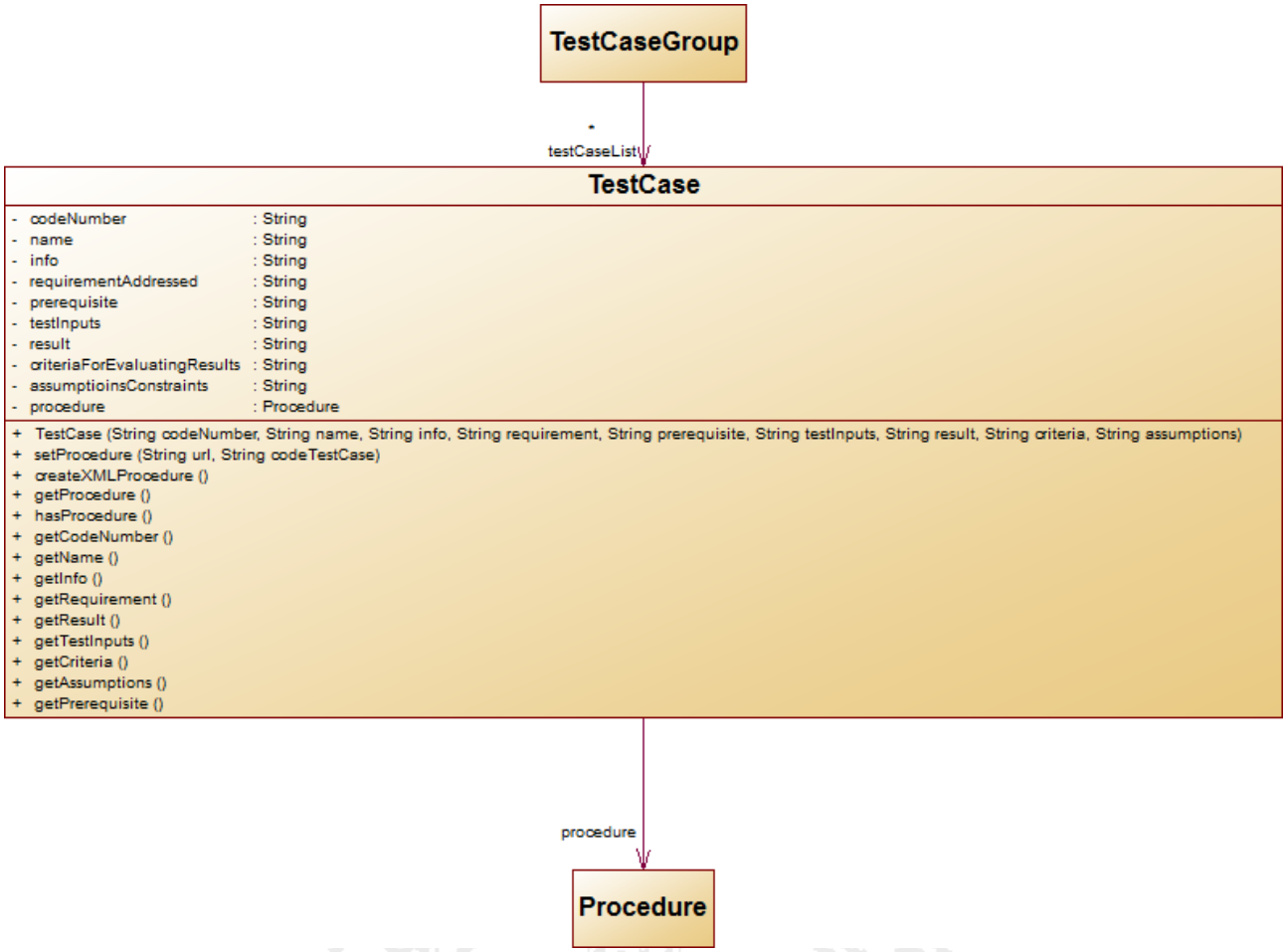

Figura 16 *TestCase UML Class description*.

La classe "*TestCase*", in Figura 16, presenta i seguenti attributi:

- *codeNumber* è un attributo che associa al test case un valore numerico, che associato al codice ereditato dal Test Suite permette di identificarlo univocamente all'interno del Progetto.
- *name* è il nome mnemonico da associare al *Test Case*;
- *info* tale parametro permette di associare al *Test Case* una descrizione;
- *requirementAddressed* è un parametro che permette di indicare i requisiti su cui il *Test Case* andrà ad interessare;
- *prerequisiti* indica quali siano i prerequisiti del caso di test;

- *testInputs* indica quali siano gli ingressi da dover fornire;
- *result* sono i risultati attesi del caso di test;
- *criteriaForEvaluatingResult* indica quali siano stati i criteri utilizzati nella scelta del caso di test;
- *assumptionsConstraints* permette di indicare quali siano i vincoli e le assunzioni fatte;
- *procedure* è un riferimento alla procedura di test da associate alla classe.
- *type* permette di definire quale tipologia di *Test Case* verranno associati;
- *testCaseList* permette di tenere un riferimento a tutti i casi di test associati alla suite;
- *testCaseProjectParent* permette di tenere un riferimento al *TestCaseProject* a cui il *TestCaseGroup* è associato.

I metodi offerti da tale classe sono:

- *setProcedure* permette di assegnare una procedura di test al *TestCase*.
- *createXMLProcedure* ogni nuova procedura di test va memorizzata, tale processo viene espletato andando a generare un file XML utilizzando la classe *XMLManager*;
- *hasProcedure* è un metodo che permette di stabilire se per un determinato *TestCase* sia stata o meno già definita una procedura;
- Infine, i metodi *get* permettono l'accesso agli attributi privati della classe.

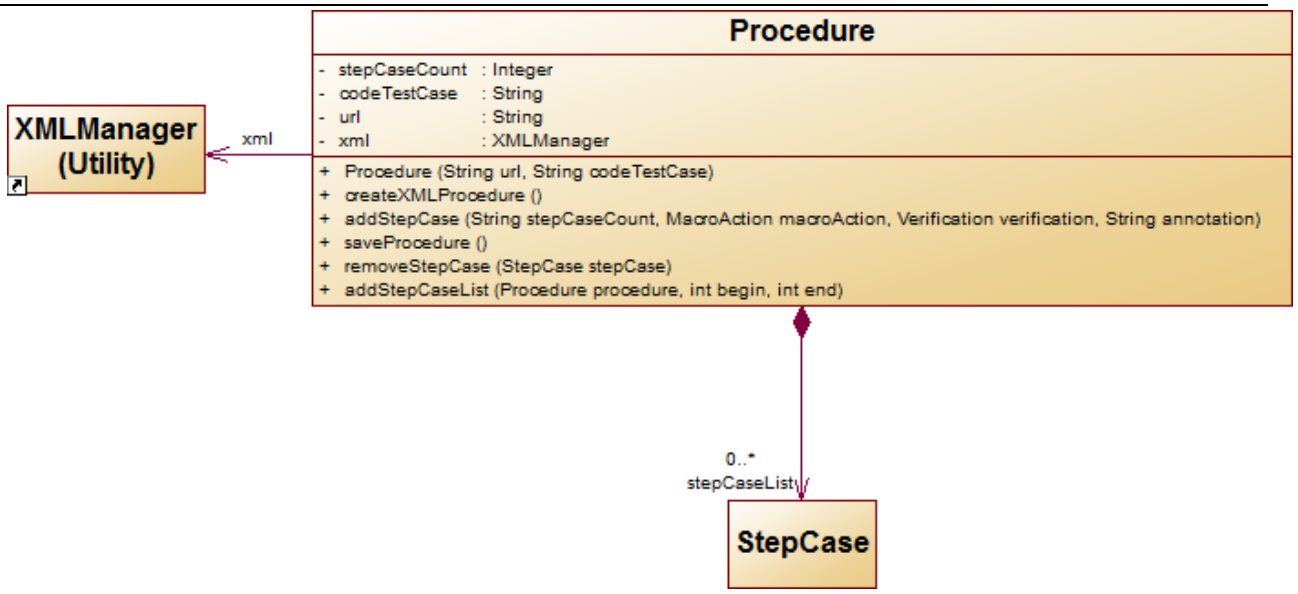

Figura 17 *Procedure UML Class description*.

La classe "*Procedure*", mostrata in Figura 17, rappresenta la procedura di test che deve essere eseguita. Ogni procedura è caratterizzata da un insieme di *stepCase*. Ogni *stepCase*, a sua volta si compone di un'insieme di azioni atomiche, denominate "*MacroAction*", da una verifica e da un'eventuale annotazione. Per cui la classe "*Procedure*" ha i seguenti parametri:

- *stepCaseCount* indica il numero corrente degli *stepCase* definiti per la procedura di test;
- *codeTestCase* permette di associare univocamente a quale *TestCase* la procedura è associata.

Per quanto riguarda i metodi offerti, essi sono:

- *createXMLProcedure* permette di creare un file *ad-hoc* in cui memorizzare l'intera procedura, per poi utilizzarla all'interno di altre procedure, ad esempio implementando eventuali precondizioni per il caso di test;
- *addStepCase* permette di aggiungere alla procedura di test un ulteriore passo sequenziale;
- *removeStepCase* permette di rimuovere lo *step case*;
- *addStepCaseList* permette di importare da un'altra procedura una sequenza di *step*.

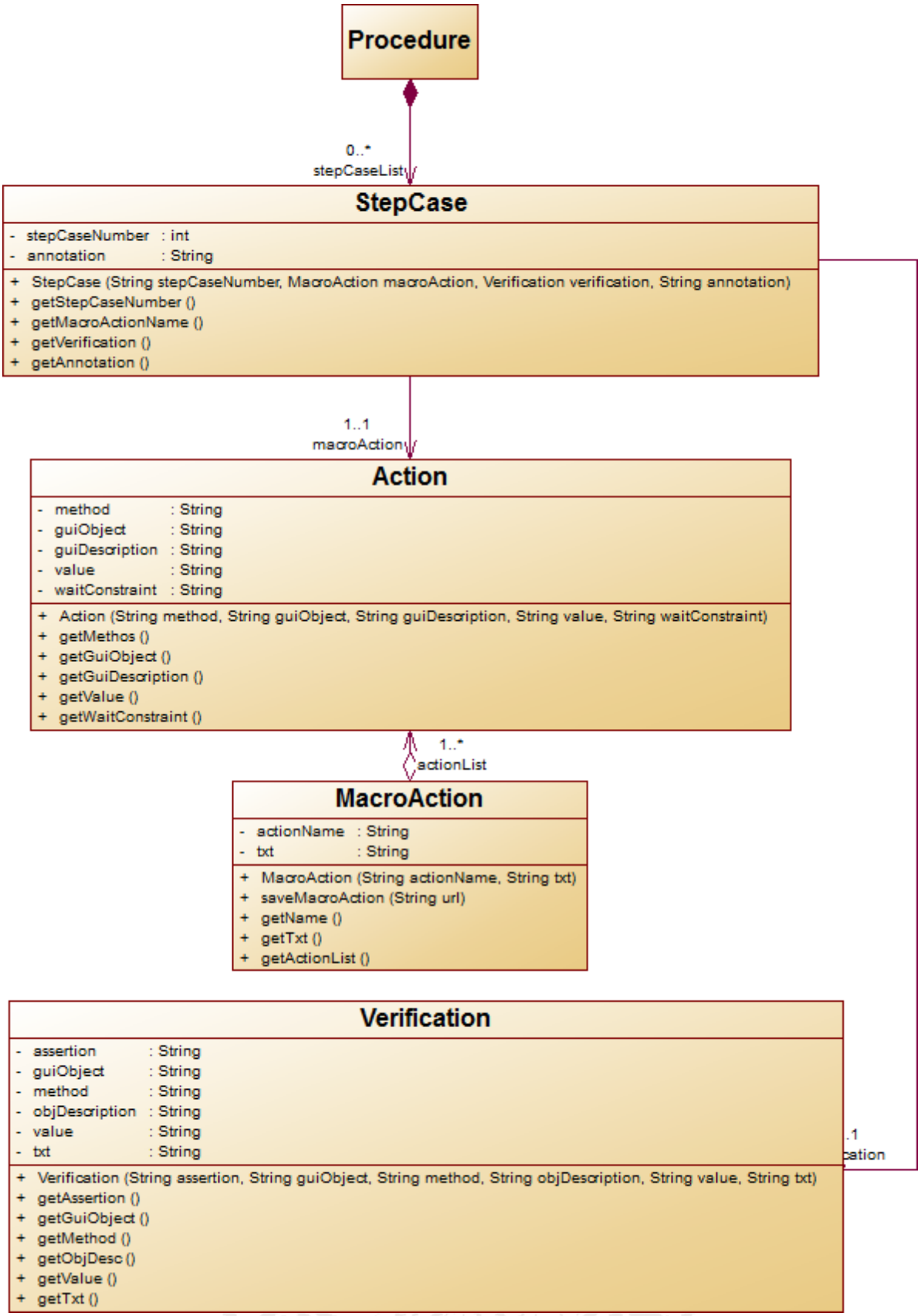

Figura 18 *StepCase UML Class description*.

In Figura 18 è mostrata la progettazione della classe *StepCase*. Tale classe rappresenta il singolo passo di una procedura. Come già accennato, ad ogni *step* vengono associate tre oggetti, di seguito elencati.

- ***MacroAction***, costituisce una sequenza di azioni atomiche. Tali azioni altro non sono che le operazioni, che un qualsiasi utente può esercitare su una qualsiasi interfaccia grafica. Di solito nella definizione delle procedure di *test* alcune sequenze di azioni si ripetono. Per tale motivo si è convenuto di creare una sorta di *repository*. In tale *repository* vengono collocate tutte le sequenze di azioni atomiche associate ad una macro azione. Grazie a ciò, tali macro azioni una volta definite, possono essere richiamate all'interno di qualsiasi procedura di *test*.
- ***Verification***, si tratta della classe che al termine di una macro azione verifica se il sistema sotto *test* mostra il comportamento atteso.
- ***Annotation*** mantiene qualsiasi tipologia di descrizione che l'utente vuole associare allo specifico *StepCase*.

Controller

Figura 19 *TesteryxController UML Class.*

La classe "*TesteryxController*", mostrata in Figura 19, riceve i comandi dall'utente, attraverso l'interfaccia utente, e li attua modificando il *data-model*. Questa classe offre diversi metodi che permettono di offrire tutte le funzionalità offerte dal *tool*. Per cui può essere considerata come classe di coordinamento dei casi d'uso. Tali metodi sono elencati di seguito:

- *newTestCaseProject* permette di creare un nuovo "*Testeryx Project*";
- *newTestCaseGroup* permette di associare al "*Testeryx Project*" una nuova *suite* di test;
- *newTestCase* permette di associare ad una *suite* un nuovo caso di test.
- *setProcedure* assegna una procedura di test già definita per un *TestCase*;
- *addStepCase* è un metodo che permette di aggiungere ad una procedura di test un ulteriore *step*;

- *updateProcedure* è utilizzato per apportare modifiche ad una procedura di test già precedentemente definita;
- *createMacroAction* permette la creazione di una nuova macro azione;
- *addAction* permette di aggiungere ad una *MacroAction* una nuova azione atomica;
- *saveMacroAction* permette di aggiungere al *repository* delle macro azioni un nuova tupla;
- *runProcedure* è il metodo che permette di avviare l'esecuzione di una procedura di test;
- *createMaveryxXML* è il metodo che permette di generare i file di configurazione del *tool "Maveryx"*, utilizzato, come già accennato, per l'esecuzione delle procedure di test.
- *loadTestCaseProject* permette di caricare un "*Testeryx Project*" precedentemente memorizzato.

Viewer

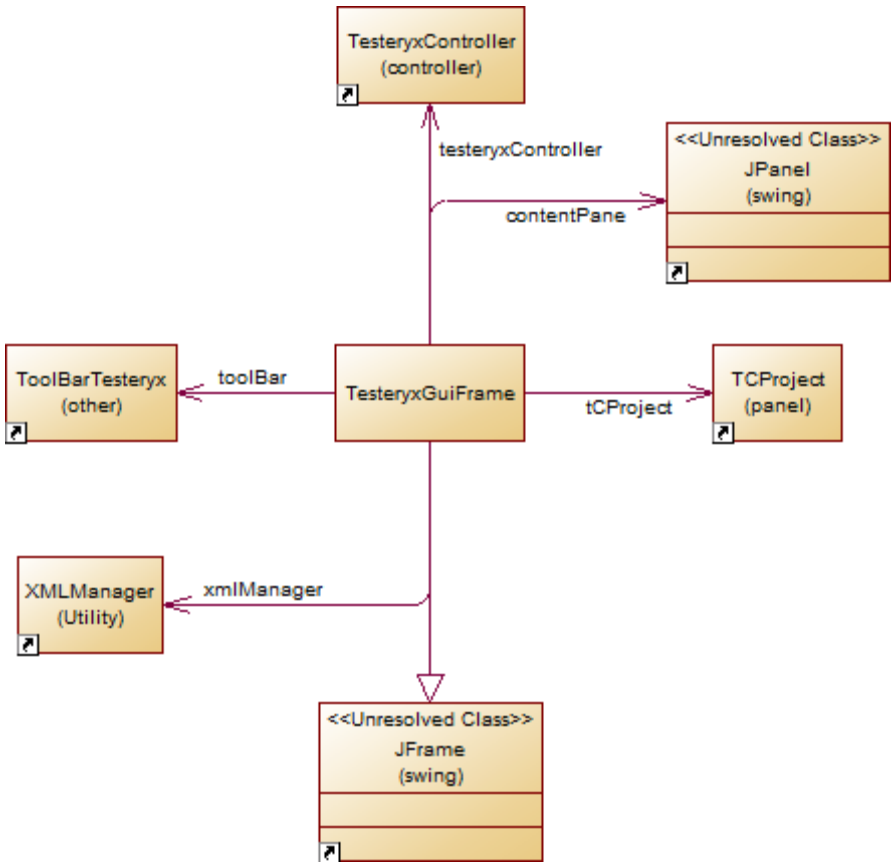

Figura 20 *View Class Diagram*.

In Figura 20 è mostrato il *class diagram* della *presentation logic*. La classe principale è "*TesteryxGuiFrame*". A tale classe vengono convogliati tutti gli eventi generati dall'utente, tramite diversi interfacce grafiche messe a disposizione da tale livello. In questo modo è possibile creare un qualsivoglia elemento grafico a cui dare il riferimento a tale classe per poter invocare i metodi da essa resi pubblici. Per semplicità si è omesso di rappresentare anche le altre classi della *presentation logic*. Una volta che l'utente ha generato un evento, esso viene inviato alla classe di controllo, la quale esegue la relativa modifica al *data-model*. Al termine di tale processo è compito del *viewer* (e quindi principalmente di tale classe), mostrare all'utente lo stato in cui ci si è portati.

Utility

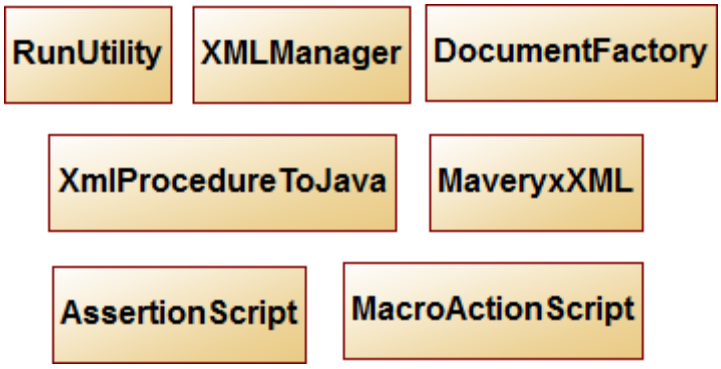

Figura 21 Panoramica delle classi di utilità.

In Figura 21 viene rappresentata una panoramica interna al *package* delle classi di utilità. Tali classi offrono diverse tipologie di servizi.

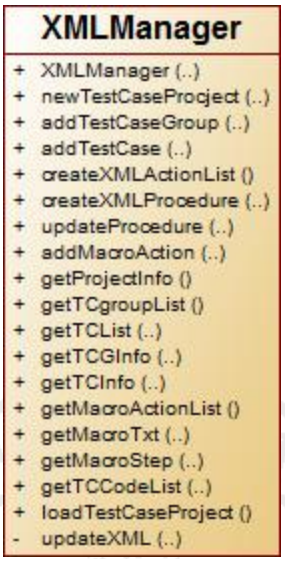

Figura 22 *XMLManager UML Class definition*.

La più importante tra le classi di utilità è "*XMLManager*", mostrata in Figura 22. Tale classe si occupa dell'accesso ad opportuni file XML. I documenti XML sono stati utilizzati per poter tenere traccia di tutte le informazioni fornite dall'utente. I metodi offerti dalla classe "XMLManager" sono i seguenti:

- *newTestCaseProject* ha il compito di istanziare il file XML avente come *root* le informazioni fornite dall'utente associate al "*Project Work*";
- *addTestCaseGroup* ha il compito di annettere un'ulteriore elemento nell'albero XML relativamente all'aggiunta di un nuovo "*Test Suite*" da parte dell'utente;

- *addTestCase* ha lo stesso compito del metodo precedente, l'unica differenza è che si riferisce all'aggiunta di un caso di test,
- *createXMLActionList* con tale metodo è possibile creare la *repository* delle macro azioni;
- *createXMLProcedure* permette di creare il file XML in cui vengono inserite tutte le informazioni inerenti ad una procedura di test;
- *updateProcedure* è il metodo che permette di memorizzare le relative modifiche apportate ad una specifica procedura di test;
- *loadTestCaseProject* permette di caricare tutte le informazioni inerenti ad un "*Testeryx Project*" già precedentemente memorizzato.
- i metodi di *get* permettono di estrapolare le informazioni contenute dei documenti XML prodotti.

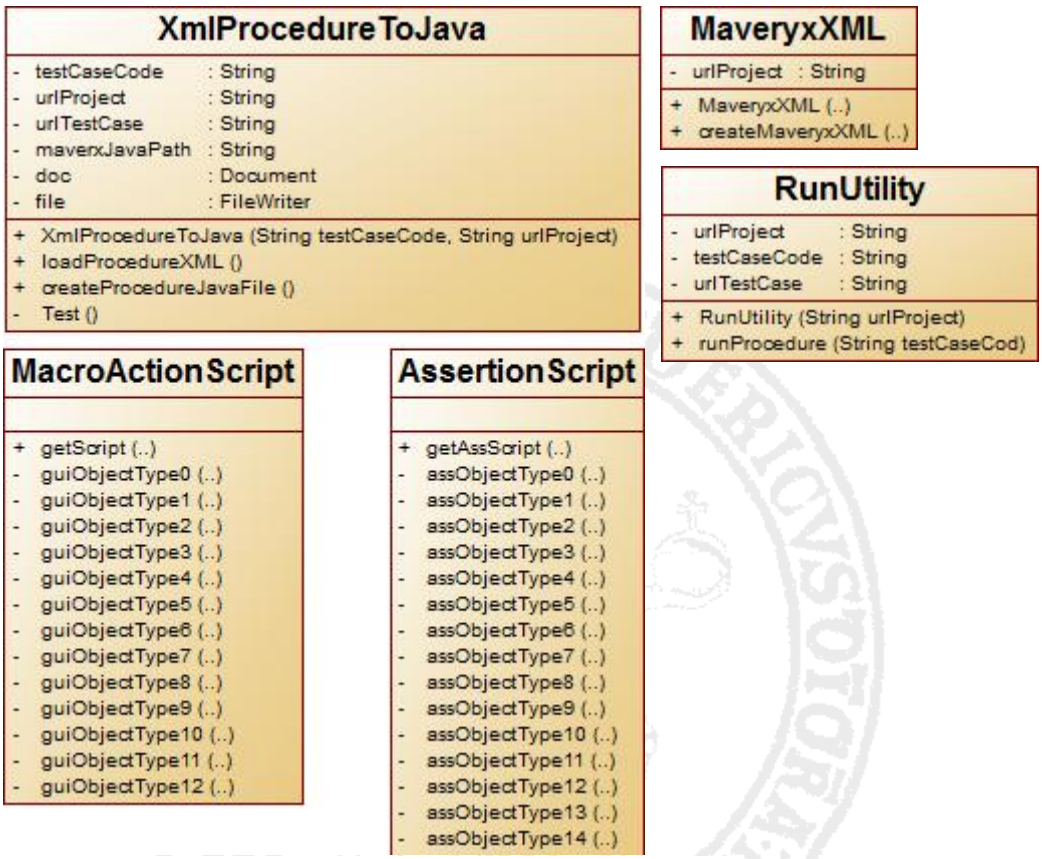

Figura 23 *Running set utility class diagram.*

In Figura 23 sono mostrate le classi di utilità che permettono l'esecuzione delle procedure di test. La classe *XMLProcedureToJava* permette la traduzione di una

procedura di test, memorizzata su di un file XML, in codice Java. Tale processo viene espletato in sinergia con altre due classi, *MacroActionScript* e *Assertion-Script*. Mediante tali classi viene effettuato il *parsing* della procedura di test in XML. Dopo aver eseguito il *parsing* viene prodotto il file Java, mediante cui dalla sua compilazione ed esecuzione è possibile avviare l'attività di *testing*. Questo compito viene assolto dalla classe "*RunUtility"*, che oltre a tale compito ha l'onere di produrre anche la relativa reportistica.

Infine, dato che lo *script* generato sfrutta le potenzialità offerte dal *tool* "*Maveryx*", quest'ultimo richiede che gli venga passato come parametro il percorso di uno specifico file XML contenete particolari informazioni. La generazione di tale documento è onere della classe "*MaveryxXML*" a cui vanno passati determinate informazioni, alcune delle quali fornite tramite un file di configurazione.

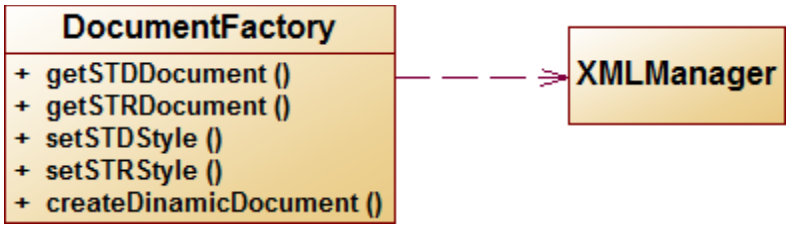

Figura 24 *DocumentFactory UML Class*.

In Figura 24 viene mostrata la classe "*DocumentFactory*", che ha il compito di generare a scelta dell'utente, la documentazione a corredo delle attività di testing svolte. I documenti prodotti sono di due tipologie: STD (*Software Testing Description*), in cui vengono descritti tutti casi di test e le relative procedure di test che dovranno essere eseguite; e STR (*Software Test Report*), in cui vengono inseriti i risultati dell'attività di *testing*.

I metodi offerti da tale classe sono i seguenti:

- *getSTDDocument* consente di poter esportare, su particolari formati di testo, le informazioni che sono di interesse di un STD, inoltre l'utente può decidere quali dati di casi di test voler esportare;
- *getSTRDocument* è il metodo analogo a quello precedente, ma riferito all'STR;

- *setSTDSTyle* permette di poter definire la formattazione di ogni tipologia di informazioni da dover esportare, tale *STDStyle* può essere memorizzato su un apposito file, in modo che una volta definiti gli stili di formattazione essi vengono automaticamente presi in considerazione;
- *setSTRStyle* è simile al metodo precedente, ma riferito ai documenti STR;
- *createDinamicDocument* permette di poter generare file testuali dinamici altamente navigabili.

3.3.5 Diagrammi sequenziali

Per passare alla definizione dell'architettura dell'applicazione sono stati realizzati dei diagrammi sequenziali, che illustrano le interazione tra l'utente e il sistema. Nei seguenti diagrammi il *Viewer* altro non è che l'insieme delle classi che compongono l'interfaccia utente.

Creazione di un "Testeryx Project"

Nel diagramma mostrato in Figura 25 viene vengono descritte la sequenza di azioni compiute nel poter creare un nuovo progetto.

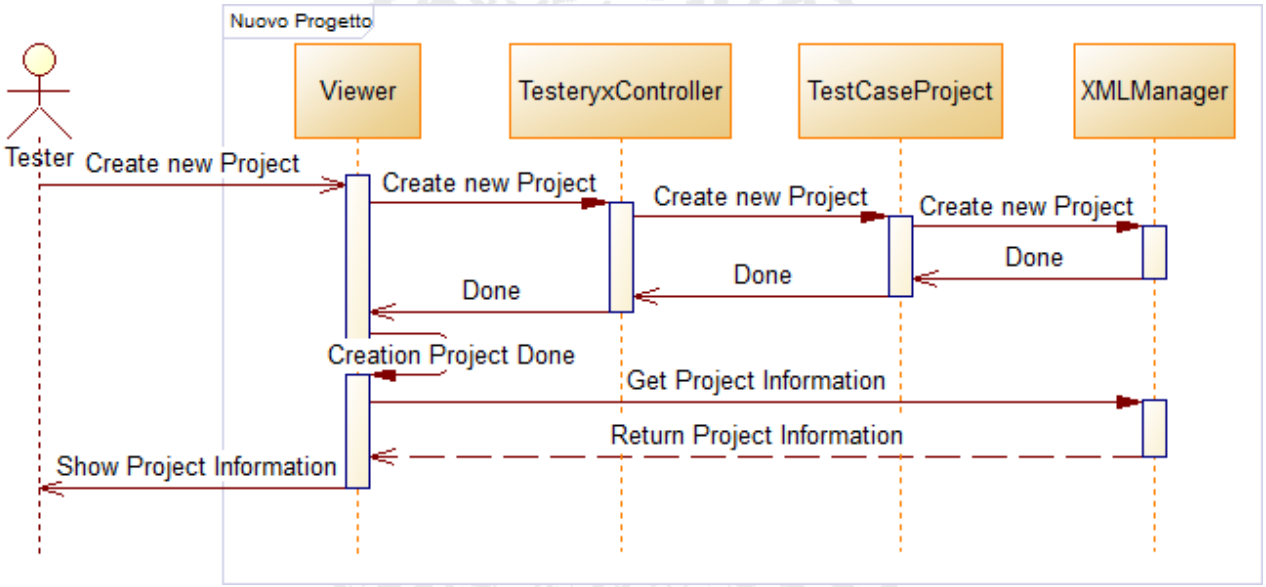

Figura 25 *Sequence Diagram* della creazione di un "*Testeryx Project*".

La creazione di uno nuovo progetto avviene nel seguente modo:

- il "*Tester*" sceglie di voler creare un nuovo progetto;
- il "*Viewer*" visualizza al "*Tester*"una schermata in cui poter inserire tutte le informazioni relative al progetto, quali: nome progetto, descrizione, percorso dove collocare il progetto da creare;
- il "*Tester*"inserisce le informazioni desiderate e avvia l'esecuzione della richiesta;
- il "*Viewer*"invoca il metodo "*createNewProject*()" della classe "*Controller*", passandogli tali informazioni;
- il "*Controller*" procede con l'inizializzazione della classe "*TestCaseProject*" invocandone il costruttore e passandone le informazioni fornite dal "Tester";
- la classe "*TestCaseProject*", a sua volta, invoca sulla classe "*XMLManager*" il metodo "*createNewProject*()";
- la classe "*XMLManager*" controlla i valori forniti e provvede alla creazione di opportuni file, se tutto va viene inviato il messaggio di "*done*";
- il "*Viewer*", ricevuto il messaggio di "*done*", ripercorrendo il percorso inverso, invoca sulla classe "*XMLManager*" il retrive delle informazioni di progetto;
- "*XMLManager*" ritorna le informazioni di progetto al "*Viewer*";
- il "*Viewer*"mostra al "*Tester*" una schermata recante le informazioni di progetto.

Aggiunta di un Test Suite

Nella Figura sottostante, viene mostrato il *sequence diagram* dell'aggiunta di una *suite di test* ad un progetto. Si parte dal presupposto che il sistema abbia caricato il progetto a cui aggiungere la suite di test.

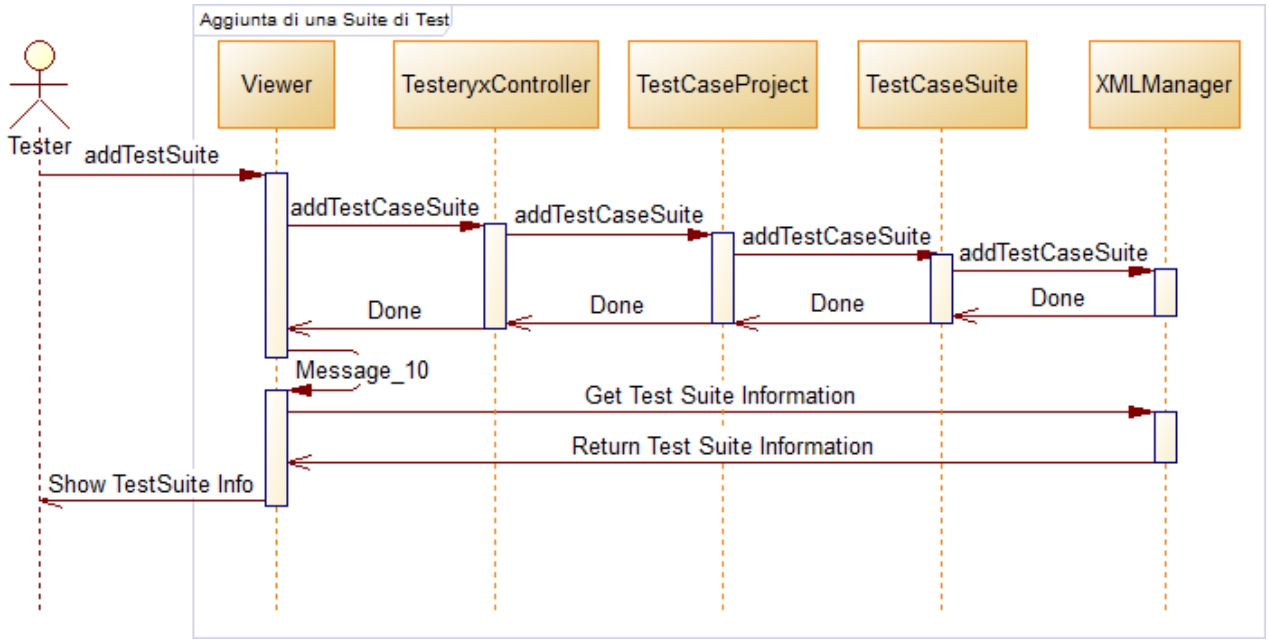

Figura 26 *Add Test Suite Sequence Diagram*.

L'aggiunta di un nuova *suite* di test avviene nel seguente modo:

- il "*Tester*", dall'apposita schermata, decide di aggiungere un nuovo *test suite* al progetto già aperto;
- il "*Viewer*" mostra una finestra in cui immettere diverse informazioni, quali: codice, nome, tipologia, descrizione;
- il "*Tester*" inserisce le suddette informazioni e invia la richiesta di aggiunta;
- il "*Viewer*" trasferisce tali informazioni al "*TesteryxController*" invocando il relativo metodo;
- il "*TesteryxController*", a sua volta, invoca sulla classe "TestCaseProject" il metodo *addTestCaseSuite*, passando le informazioni fornite dall'utente;
- la classe "*TestCaseProject*" istanzia un oggetto di tipo "*TestCaseSuite*" a cui assegna le informazioni fornite dal "*Tester*", e aggiunge un riferimento a tale

oggetto;

- la classe "*TestCaseSuite*" aggiunge un nuovo elemento alla *root* dell'albero XML, invocando il relativo metodo della classe "*XMLManager*";
- "*XMLManager*" conferma il buon fine delle operazioni, tale messaggio viene replicato a ritroso fino al "*Viewer*;
- il "*Viewer*" invoca un "*getTestSuiteInformation*" sulla classe "*XMLManager*";
- "*XMLManger* "ritorna le informazioni relative al "*TestSuite*" desiderato.

Aggiunta di un caso di test

In Figura 27 è mostrato lo scenario in cui l'utente (Tester) desidera aggiungere e definire un nuovo caso di test con annessa procedura di test.

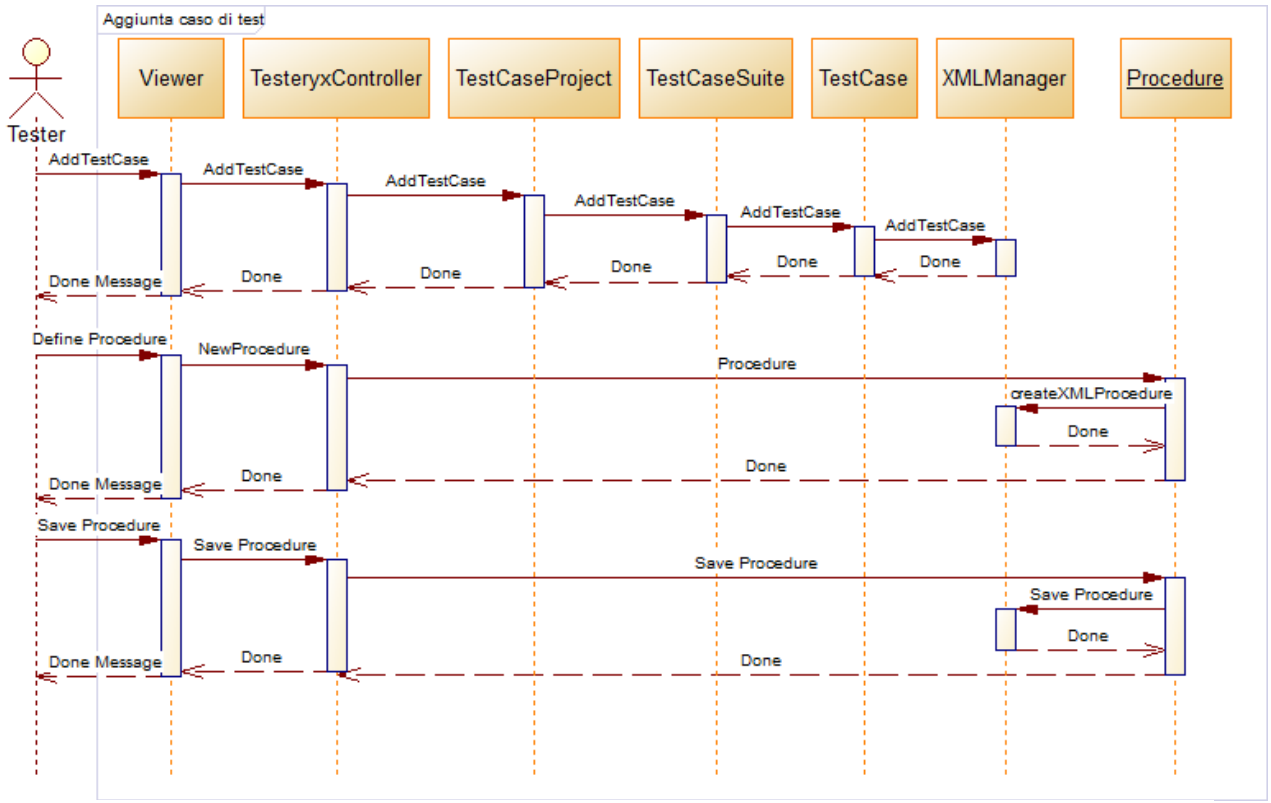

Figura 27 *Add test case sequence diagram*.

Lo scenario mostrato presenta le seguenti interazioni:

- il "*Tester*" seleziona, da un'apposita schermata della GUI (ossia il "*Viewer*"), il caso di test da dover realizzare. In tale schermata inserisce diverse infor-

mazioni, tra cui a quale suite di test associare il caso di test.

- Il "*Viewer*" invoca il metodo *addTestCase* al "*TesteryxController*";
- "*TesteryxController*" invoca il metodo *addTestCase* tale richiesta al "*TestCaseProject*";
- il "TestCaseProject" invoca il metodo *addTestCase* sulla classe "*TestCaseGroup*";
- la classe "*TestCaseGroup*" invoca il costruttore della classe "*TestCase*";
- dopodichè un messaggio di avvenuta aggiunta viene replicato a ritroso fino al "*Viewer*";
- il "*Viewer*" mostra al "*Tester*" l'avvenuta aggiunta del caso di test;
- il "*Tester*", dopo aver selezionato il caso di test desiderato, decide di voler associare una procedura di test;
- il "*Viewer*" invoca sulla classe "*TesteryxController*" il metodo *newProcedure*;
- la classe "*TesteryxController*", invoca il costruttore della classe "*Procedure*";
- la classe "*Procedure*" accede alla classe "*XMLManager*", chiedendo che venga creato l'XML associato alla procedura di test, che il "*Tester*" ha intenzione di definire;
- dopo aver fatto tutto ciò il "*Viewer*" risponde al "*Tester*" mostrando una schermata in cui è possibile definire la procedura di test;
- il "*Tester*", dopo aver definito tutti i passi della procedura di test, decide di salvare il tutto;
- il "*Viewer*" invoca sulla classe "*TesteryxController*" il metodo *saveProcedure*;
- la classe "*TesteryxController*" invoca il relativo metodo di salvataggio della classe "*Procedure*";
- "*Procedure*", a sua volta, accede alla classe "*XMLManager*" invocando il metodo *saveProcedure*, mediante cui viene effettuata la memorizzazione del-

la procedura di test definita dal Tester;

- la classe "*XMLManager*" risponde con un messaggio di "*done*", indicante l'avvenuta memorizzazione;
- tale messaggio viaggia a ritroso fino al "*Viewer*";
- il "*Viewer*" mostra un messaggio all'utente indicante l'avvenuta memorizzazione della procedura di test definita.

Avvia procedura di test

In questo prossimo diagramma vengono mostrate le sequenze di interazione, che permettono al fruitore del sistema di poter avviare l'esecuzione della procedura di test e di visualizzare l'esito.

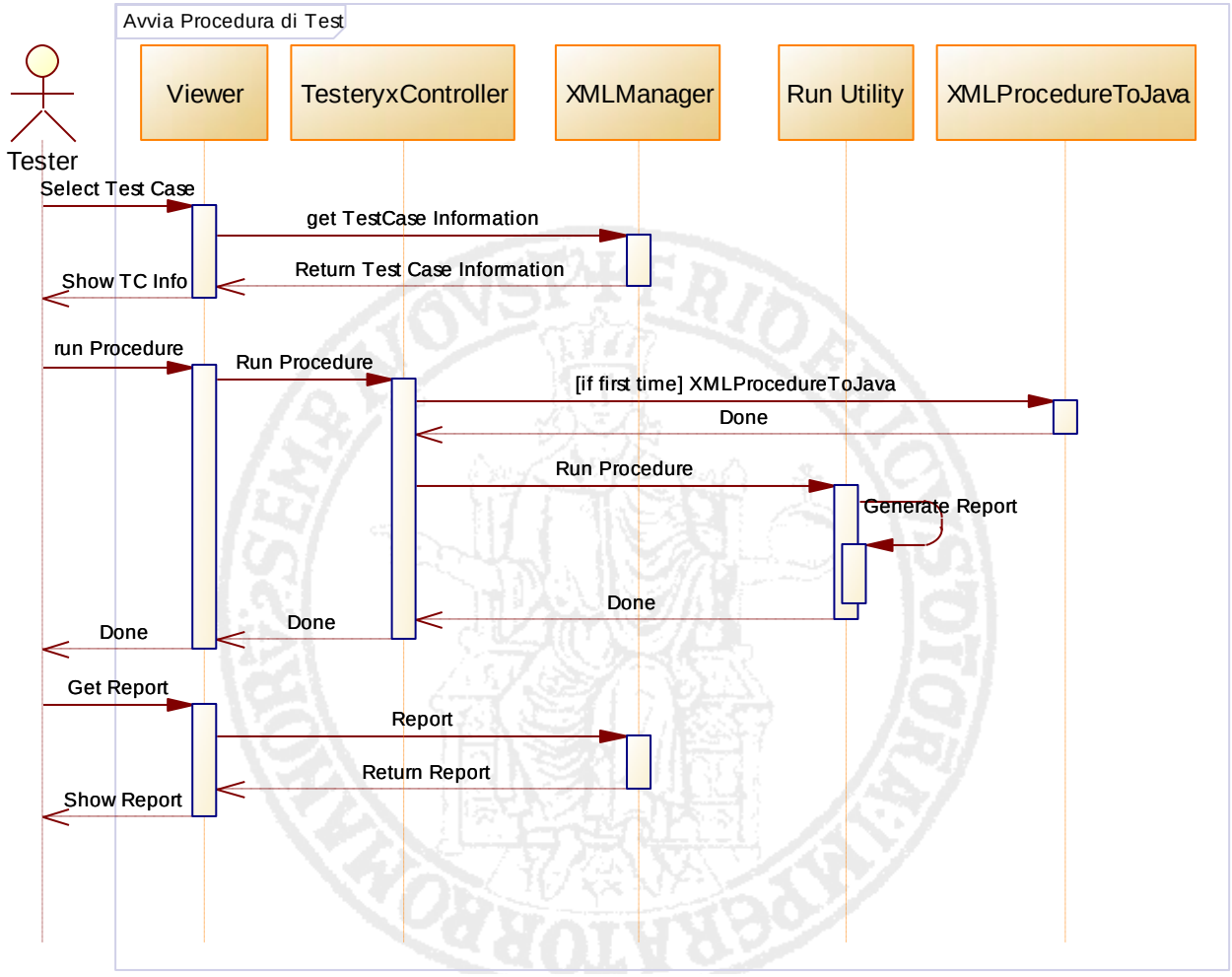

Figura 28 sequenze di interazioni per l'esecuzione di una procedura di test

Lo scenario mostrato in Figura 28 presenta le seguenti interazioni:

- Il "*Tester*" dall'interfaccia utente seleziona il caso di test desiderato;
- il "*Viewer*" accede alla classe "*XMLManager*" invocando il metodo *getTestCaseInformation*();
- la classe "*XMLManager*" si occupa di estrapolare i dati richiesti e di inviarli come risposta alla richiesta del "*Viewer*";
- il "*Viewer*" mostra all'utente le informazioni richieste;
- il "*Tester*", interagendo con il "*Viewer*", invia la richiesta di esecuzione della procedura di test associata al caso di test ottenuto in precedenza;
- il "*Viewer*", raccogliendo la richiesta del "*Tester*", invia la relativa richiesta al "*TesteryxController*";
- il " *TesteryxController* " controlla se è la prima volta che viene effettuata tale richiesta per quella determinata procedura di test. Se è così, allora tale classe invoca l'intervento della classe "*XMLProcedureToJava*" per fare generare il relativo codice, altrimenti si passa direttamente al passo successivo;
- il "*TesteryxController*" invoca la classe "*RunUtility*" per chiedere l'esecuzione della procedura di test designata;
- la classe "*RunUtility*" avvia l'esecuzione della procedura di test, e al termine genera un file di log contenente i risultati osservati;
- al termine dell'intero processo il "*Viewer*" mostra all'utente un messaggio indicante tale evento;
- il "*Tester*" invia al "*Viewer*" la richiesta di prendere visione della reportistica prodotta;
- il "*Viewer*", accede alla classe "*XMLManager*" per ottenere le informazioni richieste, per poi mostrarle al "*Tester*";

3.4 Casi di test

Nel seguente paragrafo, per verificare l'aderenza del software realizzato con le specifiche richieste, vengono mostrati alcuni dei casi di test da condurre, una volta sviluppata l'applicazione.

3.4.1 Creazione di un Project Work

Obiettivo del test è verificare che il software realizzato permetta la creazione di un "*Project Work*" in cui memorizzare e gestire tutti i dati e le informazioni inerenti alle attività di *testing*.

Requisiti indirizzati

Requisito RF1 del paragrafo 3.1.

Precondizioni

Il *tool* ***Testeryx*** è in esecuzione.

Ingressi

Non assegnati.

Risultati attesi

Creazione del "*Project Work*".

Criteri di valutazione del risultato

Osservazione del comportamento dell'interfaccia grafica.

Ipotesi e vincoli

Non assegnati.

Procedura di prova

Tabella 7 Procedura di prova creazione di un "*Project Work*".

Step	Azioni	Verifica
1	L'utente clicca sulla voce "*File*" della barra del menù e seleziona "*new Project"*.	Il sistema mostra una finestra in cui immettere le informazioni necessaria alla creazione del "*Project Work*".
2	L'utente, nella finestra mostratagli, inserisce il nome del progetto, la descrizione e il percorso in cui memorizzare il progetto. Infine, clicca sul pulsante "OK".	Il sistema crea il progetto richiesto e mostra un nuovo pannello che rappresenta l'ambiente di lavoro.

3.4.2 Gestione interna del "Project Work"

Scopo del test è verificare che il sistema permette di aggregare i casi di test in *suite di test*, in modo da poter gestire l'organizzazione interna del "*Project Work*".

Requisiti indirizzati

Requisito RF2 del paragrafo 3.1.

Precondizioni

Il *tool* **Testeryx** è in esecuzione e il "*Project Work*" è stato già creato o caricato.

Ingressi

Non assegnati.

Risultati attesi

Aggiunta ed associazione di un caso di test a una *suite*.

Criteri di valutazione del risultato

Osservazione del comportamento dell'interfaccia grafica.

Ipotesi e vincoli

Non assegnati.

Procedura di prova

Tabella 8 Procedura di prova della gestione interna del "*Project Work*".

Step	Azioni	Verifica
1	L'utente clicca dalla *toolbar* il pulsante per l'aggiunta di una *suite* di test al "*Project Work*".	Il sistema mostra una finestra in cui immettere le informazioni necessarie all'aggiunta di una *suite* di test.
2	L'utente, nella finestra mostratagli, inserisce: il codice e il nome della suite di test; la descrizione e il tipo di aggregazione. Infine, clicca sul pulsante "OK".	Il sistema aggiunge al "*Project Work*" una nuova *suite* di test, così come definita dall'utente.
3	L'utente clicca dalla *toolbar* il pulsante per l'aggiunta di un caso di test.	Il sistema mostra una finestra in cui immettere le informazioni necessarie all'aggiunta di un caso di test.
4	L'utente dopo aver inserito le informazioni desiderate (quali, Codice e Nome del caso di test, descrizione, precondizioni, ingressi, risultati attesi, criteri di valutazione del risultato, ipotesi e vincoli) e scelto a quale *suite* di test associare	Il sistema aggiunge al "*Project Work*" l'elemento relativo al caso di test.

	il caso di test definito, clicca sul pulsante "OK"	

3.4.3 Definizione e salvataggio di una procedura di test

L'obiettivo di questo caso di test è verificare che il sistema riesca a definire procedura di test e a memorizzarla in maniera persistente.

Requisiti indirizzati

Requisiti RF3 e RF4 del paragrafo 3.1.

Precondizioni

Il *tool* **Testeryx** è in esecuzione e l'utente ha già definito un caso di test.

Ingressi

Non assegnati.

Risultati attesi

Creazione e salvataggio di una procedura di test

Criteri di valutazione del risultato

Osservazione del comportamento dell'interfaccia grafica.

Ipotesi e vincoli

Non assegnati.

Procedura di prova

Tabella 9 Procedura di prova della definizione e salvataggio di una procedura di test.

Step	Azioni	Verifica
1	L'utente dal pannello relativo al caso di test desiderato, clicca sul pulsante "*Procedure*".	Il sistema visualizza una schermata in cui l'utente può definire la procedura di test.
2	L'utente definisce tutti gli "*step*" che compongono la procedura di test. Per ogni "*step*", l'utente definisce le azioni, le verifiche e le annotazioni. Al termine della composizione della procedura di prova, l'utente clicca sul pulsante "*Save*".	Il Sistema provvede alla memorizzazione persistente della procedura di test definita dall'utente. Al termine della quale visualizza all'utente un messaggio indicante l'esito della richiesta.

3.4.4 Esecuzione di un caso di test e visualizzazione report

Lo scopo di questo test è verificare che il sistema permetta l'esecuzione delle procedure di test, così come sono state definite dall'utente, e di appurare la correttezza del sistema da collaudare. Altro obiettivo del test è verificare che il sistema, ad ogni esecuzione di una procedura di test, memorizzi i risultati ottenuti.

Requisiti indirizzati

Requisiti RF5 e RF6 del paragrafo 3.1.

Precondizioni

Il *tool* **Testeryx** è in esecuzione e l'utente ha terminato di definire la procedura di test desiderata.

Ingressi

Non assegnati.

Risultati attesi

Esecuzione della procedura di test definita dall'utente e visualizzazione dei risultati ottenuti.

Criteri di valutazione del risultato

Osservazione del comportamento dell'interfaccia grafica.

Ipotesi e vincoli

Non assegnati.

Procedura di prova

Tabella 10 Procedura di prova per l'esecuzione di un caso di test e visualizzazione del *report*.

Step	Azioni	Verifica
1	L'utente, una volta definita la procedura di test, dalla schermata indicante il caso di test desiderato, clicca sul pulsante "*Run*".	Il sistema mostra all'utente una schermata in cui deve inserire alcune informazioni atte a selezionare l'applicazione da collaudare.
2	L'utente: 1. inserisce il nome dell'applicazione da testare; 2. seleziona il file ".jar" recante l'applicazione da collaudare; 3. inserisce gli argomenti da passare alla classe *main* dell'applicazione da verificare; 4. inserisce gli argomenti da passare alla *Java Virtual Machine*.	Il sistema provvede all'esecuzione della procedura di test. Al termine della quale, il sistema, visualizzerà un messaggio né indica il completamento.

	Infine l'utente clicca sul pulsante "OK".	
3	L'utente clicca sul pulsante "*View Report*" per visualizzare l'esito del processo di test.	Il sistema visualizza una schermata in cui è descritto l'esito della procedura di test. Al termine di tale processo, il sistema, visualizzerà un messaggio che ne indica il completamento.

3.4.5 Esecuzione selettiva delle procedure di test già definite

Obiettivo del test è verificare che il sistema permette di avviare l'esecuzione selettiva di procedure di test già definite dall'utente.

Requisiti indirizzati

Requisito RF7 del paragrafo 3.1.

Precondizioni

Il *tool* **Testeryx** è in esecuzione e l'utente ha definito alcune procedure di test riconducibili alla stessa suite.

Ingressi

Non assegnati.

Risultati attesi

Esecuzione sequenziale automatica delle procedure di test selezionate.

Criteri di valutazione del risultato

Osservazione del comportamento dell'interfaccia grafica.

Ipotesi e vincoli

Non assegnati.

Procedura di prova

Tabella 11 Procedura di prova dell'esecuzione selettiva delle procedura di test già definite.

Step	Azioni	Verifica
1	L'utente seleziona dall'albero del progetto il nodo relativo alla suite di test desiderata.	Il sistema mostra all'utente, il pannello in cui sono contenute tutte le informazioni associate alla suite di test selezionata.
2	L'utente clicca sul pulsante "*Run*".	Il sistema visualizza una finestra riportante tutte le procedure di test già definite dall'utente associate alla suite desiderata.
3	L'utente effettua una selezione multipla delle procedure di test che si desidera avviare.	Il sistema provvede all'esecuzione sequenziale delle procedure di test selezionate.

3.4.6 Generazione automatica della documentazione

L'obiettivo di questo caso di test è verificare che il sistema riesca a generare, dalle informazioni fornite dall'utente e dalle attività svolte, i documenti a corredo del processo di *testing* quali STD e STR.

Requisiti indirizzati

Requisito RF8 del paragrafo 3.1.

Precondizioni

Il *tool* **Testeryx** è in esecuzione.

Ingressi

Non assegnati.

Risultati attesi

Creazione e salvataggio di una procedura di test

Criteri di valutazione del risultato

Osservazione del comportamento dell'interfaccia grafica.

Ipotesi e vincoli

Non assegnati.

Procedura di prova

Tabella 12 Procedura di test della generazione automatica della documentazione.

Step	Azioni	Verifica
1	L'utente clicca sulla voce "*File*" della barra del menù e seleziona "*load Project"*	Il sistema mostra una finestra in cui immettere le informazioni necessaria al caricamento del progetto desiderato.
2	L'utente, nella finestra mostratagli, inserisce il percorso in cui è stato salvato il progetto desiderato, dopo di ché clicca sul pulsante "OK".	Il sistema carica il progetto richiesto e mostra un nuovo pannello che rappresenta l'ambiente di lavoro.
3	L'utente dalla "*toolBar*" clicca sul pulsante "*export documentation*".	Il sistema mostra una finestra in cui l'utente deve inserire alcune informazioni.
4	L'utente seleziona quale documento generare (ovvero STD o STR o entrambi) e quale casi di test riportare nella documentazione.	Il sistema genera la documentazione richiesta, al termine della quale, mostra un messaggio che né indica il completamento.

Capitolo 4
Sperimentazione

In questo capitolo si procede ad una prima reale sperimentazione di **Testeryx**. In questo modo ci si può rendere conto di quali siano realmente i suoi punti di forza e su quali aspetti occorre lavorare per migliorare il sistema nelle successive *release*.

4.1 Caso di studio

Il caso di studio preso in esame è classificato da **SELEX S.I.** come altamente riservato, pertanto, ad alcune entità e funzionalità di seguito analizzate e descritte sono state assegnate dei nomi fittizi.

4.1.1 Sistema "Radar controfuoco"

SELEX Sistemi Integrati, una società **Finmeccanica**, si è aggiudicata, in qualità di team leader dell'associazione temporanea di imprese con la società SAAB Microwave SA, una gara del valore di 83 milioni di euro per la fornitura al Ministero della Difesa – Teledife (Direzione Generale delle Telecomunicazioni dell'Informatica e delle Tecnologie Avanzate) di cinque sistemi radar controfuoco e del relativo supporto logistico.

L'intera fornitura sarà completata entro 36 mesi dall'entrata in vigore del contratto che prevede, anche, attività didattiche di formazione agli operatori e il

supporto all'installazione e alla gestione del sistema.

SELEX Sistemi Integrati, in qualità di responsabile della commessa e grazie alle competenze come integratore di sistemi, avrà il compito di gestire l'intero progetto. L'azienda si occuperà infatti dell'integrazione dei sistemi di comando e controllo SIACCON, precedentemente forniti all'Esercito italiano, con i radar Arthur della SAAB Microwave. **SELEX Sistemi Integrati** provvederà, inoltre, alla modifica degli apparati radar e alla fornitura di tutti gli equipaggiamenti di navigazione e comunicazione, nonché, alla mobilità del sistema.

I radar controfuoco sono sistemi per la sorveglianza del campo di battaglia che consentono la protezione delle unità sul terreno operativo. I sistemi sono infatti in grado di localizzare le sorgenti di fuoco avversarie e rispondere in modo tempestivo, anche, in avverse condizioni ambientali, grazie alla facilità di rilevamento dei dati, di dispiegamento e di mobilità sul campo. Tutte le informazioni raccolte vengono inviate ad un centro di comando e controllo dedicato alla gestione della missione. I sistemi possono, inoltre, cooperare con velivoli non pilotati (UAV) ed essere integrati con i sistemi di comando e controllo della NATO.

I sistemi controfuoco forniti da **SELEX Sistemi Integrati** rispondono alle crescenti responsabilità che l'Amministrazione Difesa italiana ha assunto negli ultimi anni nel contesto internazionale. La fornitura è, infatti, destinata a missioni fuori area e consentirà alle Forze Armate italiane di operare in maggiore sicurezza.

Nel contesto della sorveglianza del capo di battaglia un radar è utilizzato per:

- localizzare, nel settore assegnato, sorgenti di fuoco attivo avversarie, con identificazione del punto di impatto e classificazione delle sorgenti, fornendo

tutti i dati necessari per consentire l'intervento delle artiglierie amiche;

- osservare il fuoco delle artiglierie amiche, al fine di stimare lo scarto tra l'obiettivo e il punto di impatto del proiettile, fornendo tutti i dati necessari all'eventuale aggiustamento del tiro.

La figura sottostante rappresenta visivamente le funzionalità appena descritte.

Figura 29 Schema di impiego del sistema Radar Controfuoco.

Il radar controfuoco svolge la sua missione coordinandosi con un centro superiore di Comando e Controllo, da cui riceve la missione e tutti i dati necessari alla corretta rilevazione (meteo, per esempio) e le batterie, per poi comunicare con rapidità l'informazione elaborata.

Tipico è altresì l'impiego di più di un sensore in una stessa Area di Interesse, al fine di garantire una migliore copertura e adattabilità all'evoluzione dello scenario tattico.

Da quanto detto scaturisce come tempestività, manovrabilità e l'interoperabilità con tutti i soggetti che operano sullo scenario operativo, siano requisiti fondamentali per il corretto impiego del sistema.

Considerato che le nostre FF.AA. sono sempre più spesso chiamate ad operare in missioni multinazionali o di coalizione, nel progetto del RCF proposto, si è dedicata speciale attenzione a garantire la più larga compatibilità del sistema con gli standard e i formati in uso nei contesti operativi nazionali di Coalizione multinazionale e NATO [34].

4.1.2 CSCI SAN

Il CSCI (*Computer Software Configuration Item*) SAN è usato nel sistema Radar Controfuoco. È il componente di interfaccia verso i sistemi esterni. Il SAN fornisce una rappresentazione a oggetti di tutti i flussi informativi previsti dalle interfacce. Esso è integrato facilmente e in modo disaccoppiato con il gestore dei *workflow* dei servizi della digitalizzazione. Gli oggetti sono condivisi e usati per le logiche applicative previste per il sistema. Inoltre, si prevede la sua integrazione con altri sistemi.

Il CSCI SAN deve realizzare le seguenti funzionalità:

- verifica e validazione di messaggi appartenenti a specifici protocolli applicativi;
- rappresentazione delle informazioni contenute nei messaggi attraverso oggetti condivisibili con altri applicativi;
- pubblicazione degli oggetti attraverso un modello condiviso di scambio messaggi;
- ricezione ed invio messaggi su un dato protocollo di comunicazione;

Il SAN deve essere costituito da 3 CSU (*Component System Unit*):

- Il SEM (*System Exchange Message*): ha il compito di permettere lo scambio delle informazioni contenute nei messaggi standard fornendo una loro rappresentazione tramite opportuni *Data Trasfer Object* (DTO) e verificando la validità dei messaggi stessi.

- Il MCP (*Multiple Communication Protocol*): ha il compito di gestire i diversi protocolli di comunicazione che possono essere utilizzati per inviare o ricevere i messaggi di uno standard.
- Il MRM (*Model Remote Message*): ha il compito di gestire la pubblicazione dei messaggi provenienti dai sistemi esterni, definendo un modello condiviso di comunicazione.

I CSU devono essere basati su interfacce, in modo da consentire implementazioni separate e specifiche. In particolare devono essere progettati utilizzando pattern creazionali quali *Abstract Factory* o *Factory Method*.

I CSU devono essere progettati in modo da consentire a *runtime* il caricamento di specifiche implementazioni, in particolare, deve essere prevista la possibilità di configurare in opportuni file le implementazioni da avviare allo *startup* dei componenti.

Il CSCI SAN deve fornire interfacce di utilizzo dei propri servizi basate su *stateless bean* (*ejb*) e web *services* (*ws*).

Le operazioni esposte dall'interfaccia del SAN (metodi degli *ejb* e *service method* dei ws) devono essere raggruppate secondo funzionalità a cui afferiscono (classe di servizio):

- **Standards**: è la classe di servizio per la gestione dei messaggi di uno standard;
- **AppProtocols**: è la classe di servizio per la gestione dei differenti protocolli applicativi gestiti;
- **ServiceMessagings**: è la classe di servizio per la diffusione dei messaggi provenienti dai sistemi esterni.

Per ogni specifico servizio, ovvero per ogni relativo metodo, devono essere individuati tutti i possibili errori e gestiti in termini di eccezioni. In particolare è ne-

cessario utilizzare un modello centralizzato delle eccezioni, in grado di loggarle e ribaltarle al chiamante.

4.1.3 Informazioni generali sulla sperimentazione

Testeryx affinchè possa essere utilizzato, ha come requisito essenziale che il sistema da sottoporre a collaudo sia dotato di un un'interfaccia grafica scritta in Java.

Visto che il sistema da dover collaudare non è dotato di un'interfaccia grafica in Java, ma interagisce mediante scambio di messaggi mediante il protocollo di trasporto TCP(*Trasmission Control Protocol*), è stata creata un'applicazione che permette di inviare e ricevere messaggi TCP con il sistema da testare. L'applicazione così realizzata è stata denominata "*Socket Test*".

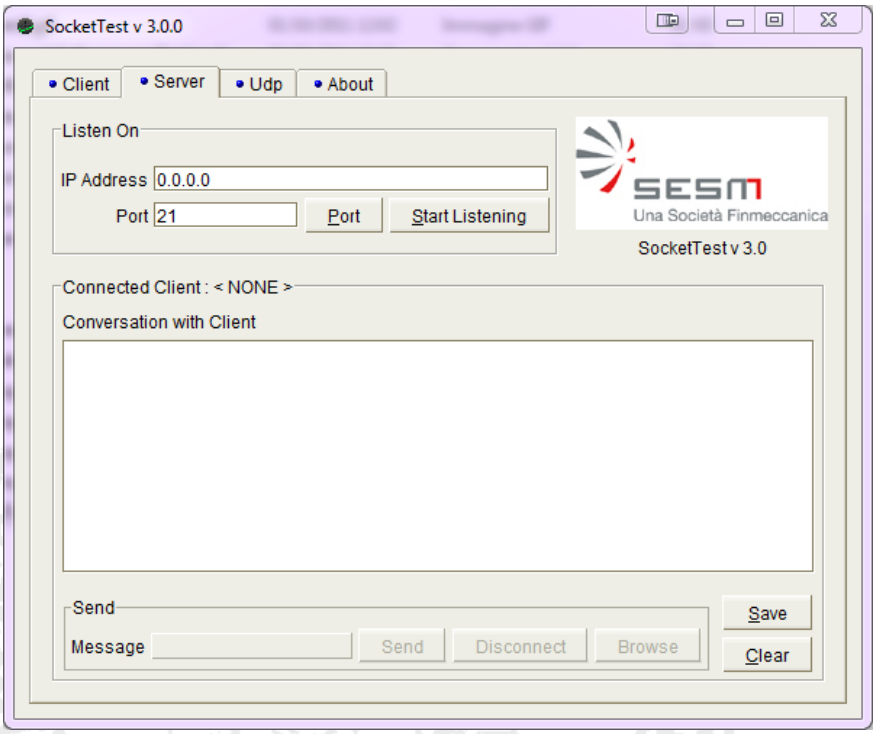

Figura 30 Interfaccia grafica Java dell'applicazione "*Socket Test*" .

Sollecitando direttamente tale applicazione, mediante le procedure di test definite con **Testeryx**, si è potuto verificare la correttezza del sistema di nostro interesse. In Figura 31 viene mostrato lo scenario in cui sono state condotte le attività di test.

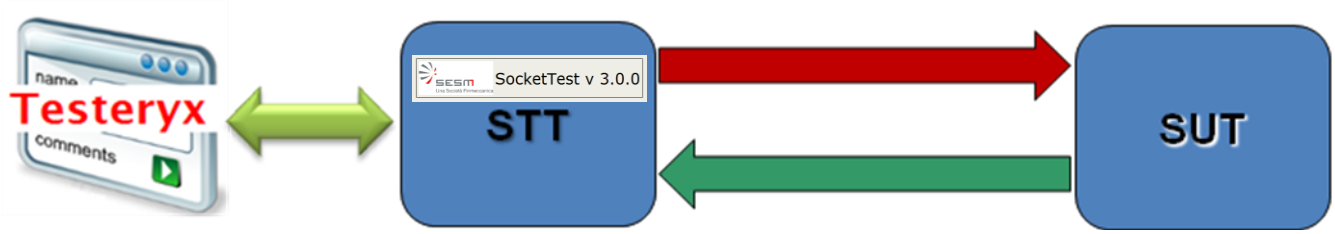

Figura 31 Scenario del processo di *testing* adottato.

4.2 Scrittura ed esecuzione dei casi di test

Di seguito vengono mostrati alcuni casi di test realizzati per verificare che il software implementato rispetti alcuni dei requisiti richiesti.

4.2.1 *Startup* della connessione

Obiettivo del test è verificare che il SAN, una volta avviato, provveda a stabilire una connessione con il *listener* configurato nel file di configurazione. In particolare, al fine di avviare una connessione TCP con il radar *Arthur* in ascolto verrà inviato il messaggio di *ready*. Il SAN provvederà, inoltre, ad inviare il messaggio di *alive* secondo quanto prestabilito.

Di seguito viene documentato ciò che è stato fatto al fine di poter comprovare tale caratteristica del sistema. In pratica è stata definita una procedura di prova, volta a verificare che una volta avviata la connessione TCP con il SAN, si riceva il messaggio di *ready*. Tale messaggio viene visualizzato in un apposito campo testuale, per cui è stata definita una procedura di test, la quale una volta stabilita la connessione con il SAN si attende la ricezione del messaggio di *ready*.

Tabella 13 Procedura eseguita per implementare il caso di test dello "*Startup* dell connessione".

Fasi	Azioni svolte
Fase iniziale	1. Il CSCI SAN è avviato 2. *Testeryx* viene avviato e: a. creato un nuovo progetto denominato "Radar Controfuoco"; b. associato al progetto un *test suite* denominato

	"SAN"; c. associato alla test suite creata un caso di test, nominato "*Startup*"; d. si accede alla schermata di definizione della procedura di test.
Scenario principale	1. Si definisce una nuova macro azione denominata "*Startup*", nella quale si instaura una connessione tra i due *end point*, mostrata in Figura 32; 2. Si definisce una verifica, la quale controlla la ricezione del messaggio di *ready*, così come mostrato in Figura 33. 3. Si compone la procedura di test, come mostrato in Figura 34. 4. Infine si clicca sull'icona "*run procedure*" e si associa il file "*SocketTestTool.jar*", mostrato in Figura 35.

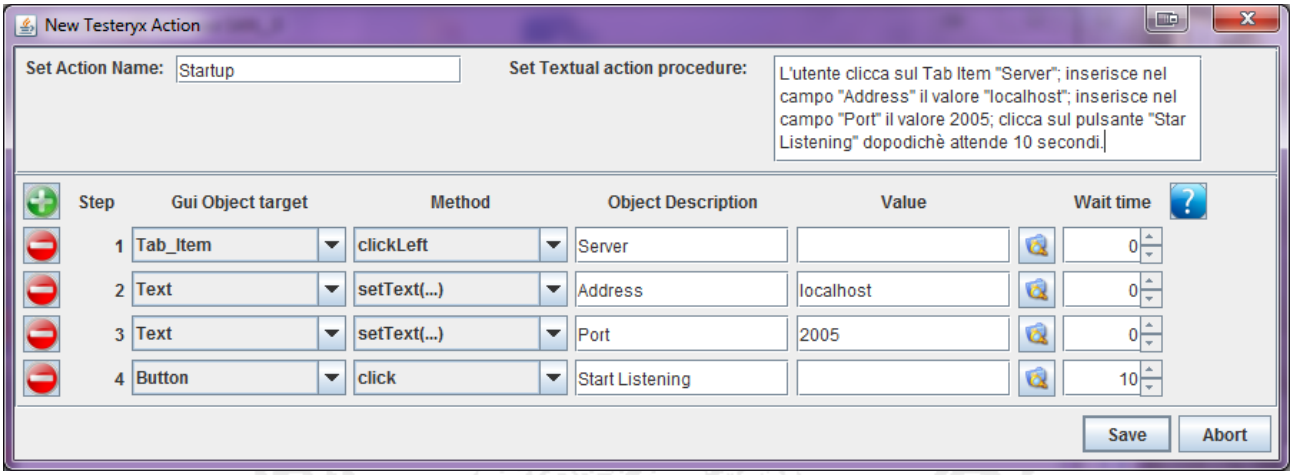

Figura 32 Definizione macro azione "*Startup*".

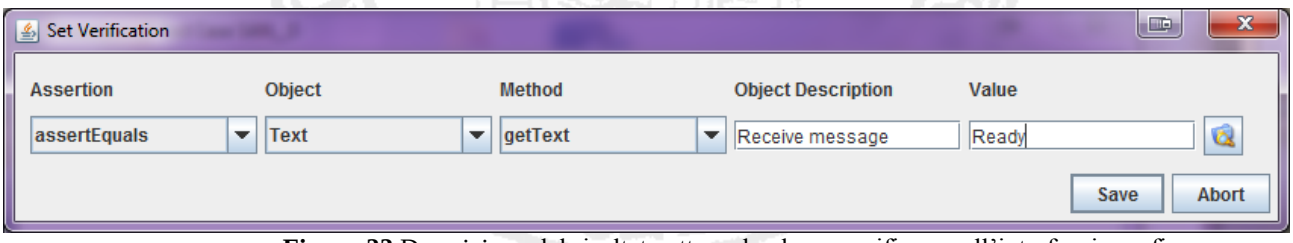

Figura 33 Descrizione del risultato atteso da dover verificare sull'interfaccia grafica.

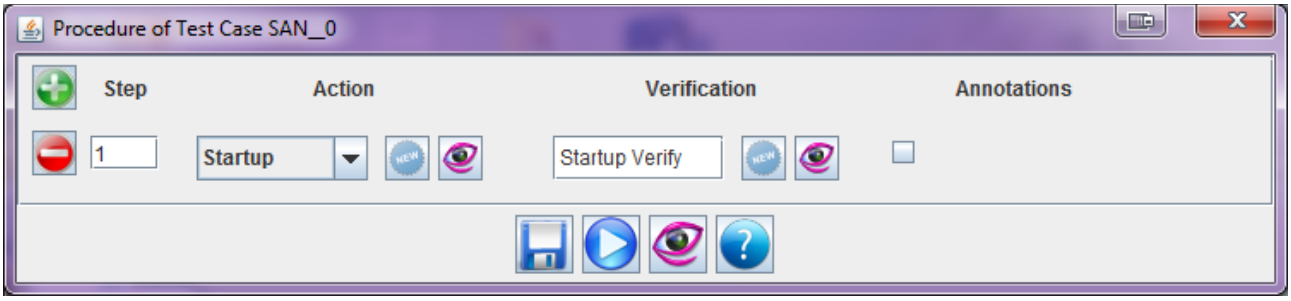

Figura 34 Procedura definita per il caso di test "Startup della connessione".

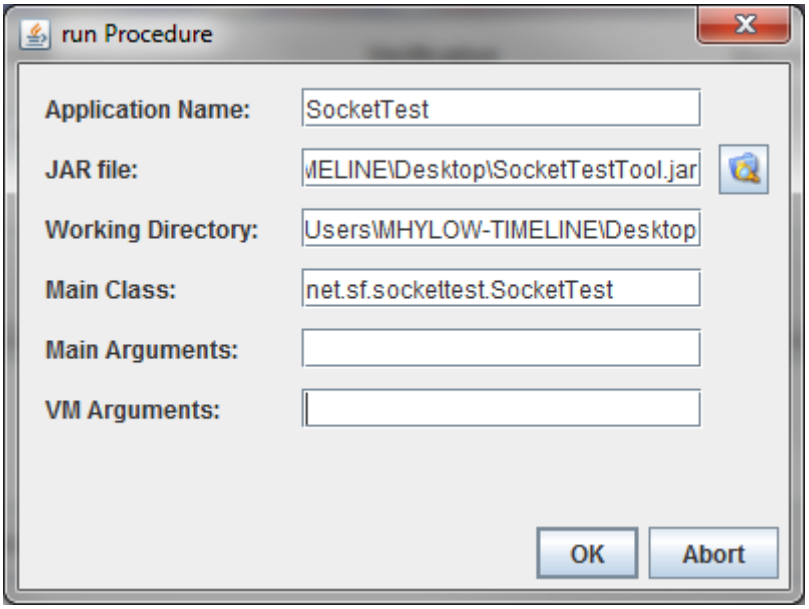

Figura 35 Associazione del file "*SocketTestTool.jar*"

4.2.2 Ricezione periodica del "*keep alive message*"

Lo scopo del test è verificare che il SAN, una volta eseguita la procedura di *start-tup*, cominci l'invio del messaggio di "*keep alive*". La cadenza di tali comandi periodici è prefissato.

Nel proseguire nella definizione della procedura di test si aggiunge un nuovo caso di test, annettendolo alla suite di test creata in precedenza. In pratica tale caso di test include anche quello precedente. Dopo aver effettuato lo *Startup* e verificato la ricezione del messaggio di *Ready* si attende la ricezione del messaggio di *keep alive* ogni *n* secondi.

Tabella 14 Procedura eseguita per implementare il caso di test della ricezione del "*Keep alive message*".

Fasi	Azioni svolte
Fase iniziale	1. Il CSCI SAN è avviato 2. *Testeryx* è avviato e: a. associato alla test suite "SAN" un caso di test denominato "*Keep Alive*"; b. si accede alla schermata di definizione della procedura di test.
Scenario principale	1. Si aggiunge al progetto una nuova macro azione, che in realtà non esegue nessuna azione precisamente ma serve a indicare un'attesa di *n* secondi; 2. Si definisce una verifica, la quale controlla la ricezione del messaggio di "*keep alive*", così come mostrato in Figura 36. 3. Si compone la procedura di test, come mostrato in Figura 37. 4. Infine si clicca sull'icona "*run procedure*" e si associa il file "*SocketTestTool.jar*", mostrato in Figura 35.

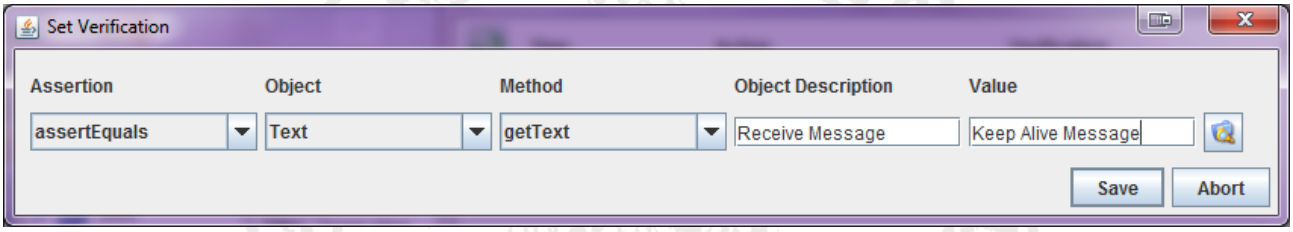

Figura 36 Descrizione della verifica dell'avvenuta ricezione del messaggio di *keep alive*.

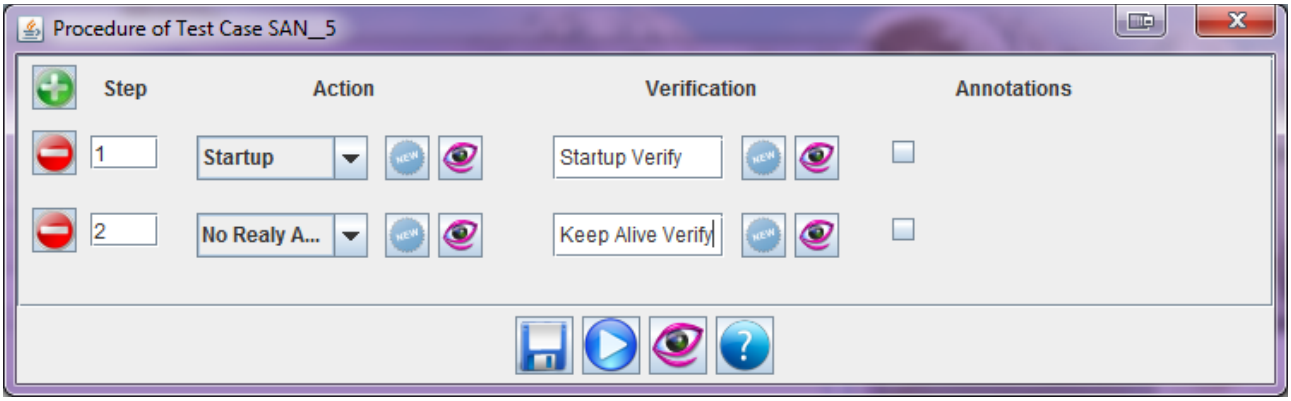

Figura 37 Composizione della procedura della ricezione del messaggio di *keep alive*.

4.2.3 Ricezione del messaggio di "*start communication*"

L'obiettivo del test è di verificare che il SAN riceva e gestisca opportunamente il messaggio di "*start comunication*".

Per la definizione della seguente procedura di test, è stato annesso alla suite di test creata in precedenza, un nuovo caso di test. In realtà questo caso di test include in sequenza anche gli altri due esaminati in precedenza.

Tabella 15 Procedura eseguita per implementare il caso di test della ricezione dello "*start comunication message*".

Fasi	Azioni svolte
Fase iniziale	1. Il CSCI SAN è avviato 2. *Testeryx* è avviato e: a. associato alla test suite "SAN" un caso di test denominato "*Start Communication*"; b. si accede alla schermata di definizione della procedura di test.
Scenario principale	1. Si aggiunge al progetto una nuova macro azione che provvede ad inviare al SAN il messaggio di "*Start Communication*", così come è mostrato in Figura 38; 2. Si definisce una verifica, la quale controlla la ricezione del relativo messaggio di *acknowledgement*, così come mostrato in Figura 39.

3. Si compone la procedura di test, come mostrato in Figura 40.
4. Infine si clicca sull'icona "*run procedure*" e si associa il file "*SocketTestTool.jar*", mostrato in Figura 35.

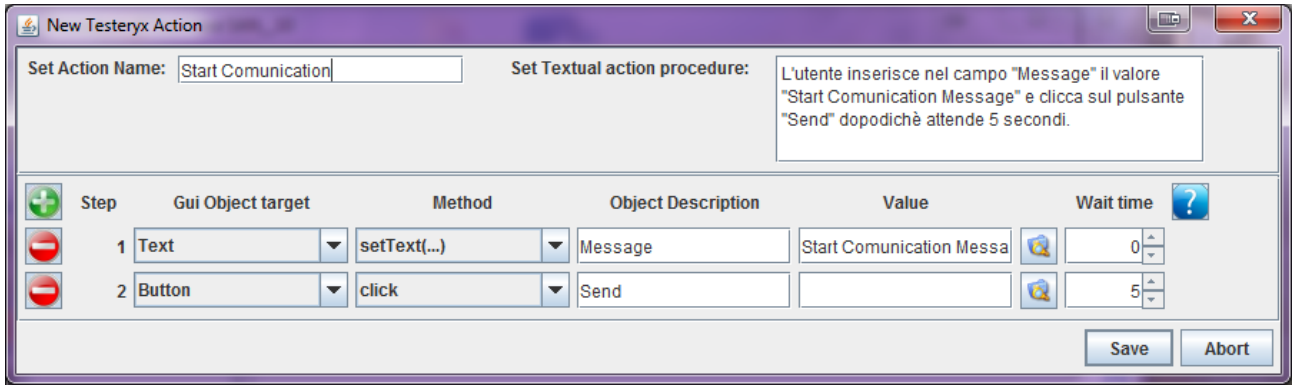

Figura 38 Definizione della macro azione "Start Comunication".

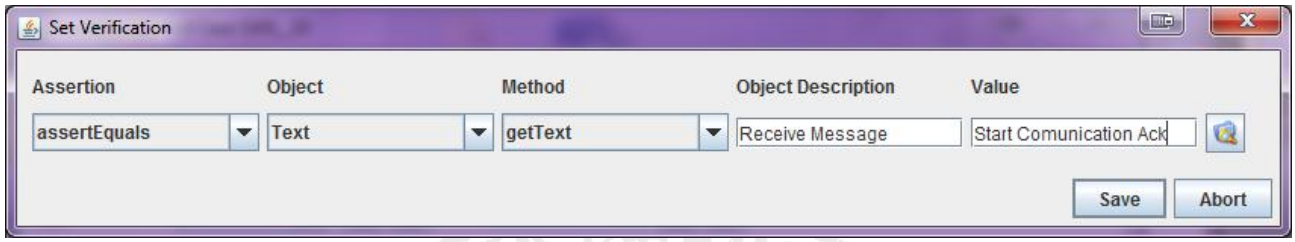

Figura 39 Verifica dell'avvenuta ricezione del messaggio di *Acknoledgement*.

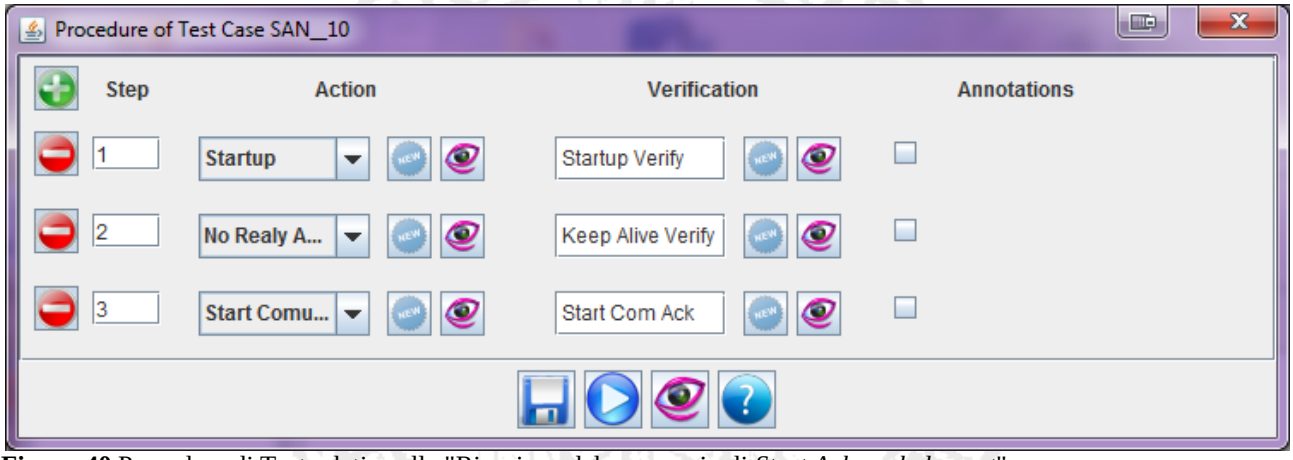

Figura 40 Procedura di Test relativa alla "Ricezione del messaggio di *Start Acknowledgment*".

4.2.4 Ricezione dei "*Transmitted message acknowledgement*"

Scopo del test è verificare la capacità del CSCI SAN di ricevere le diverse tipologie di messaggi definiti, a cui si attende la ricezione del relativo messaggio di "*acknowledgement*".

Il CSCI SAN una volta instaurata la connessione con il radar *Arthur* riceve diverse tipologie di messaggi, ogn'uno dei quali ha un predefinito messaggio di risposta. In quest'ultimo caso di test viene inviato al SAN uno dei messaggi/comandi già definiti e si attende la relativa risposta. Così come è avvenuto negli altri casi, si è aggiunto alla suite di test un nuovo *test case* a cui, poi, è stata definita una procedura. Di seguito sono mostrati i passi effettuati.

Tabella 16 Procedura eseguita per implementare il caso di test della ricezione degli *Acknowledgement*.

Fasi	Azioni svolte
Fase iniziale	1. Il CSCI SAN è avviato 2. *Testeryx* è avviato e: a. associato alla test suite "SAN" un caso di test denominato "*acknowledgement*"; b. si accede alla schermata di definizione della procedura di test.
Scenario principale	1. Si aggiunge al progetto una nuova macro azione che provvede ad inviare al SAN uno specifico messaggio, per cui denominiamo "*Send Message*", così come è mostrato in Figura 41; 2. Si definisce una verifica, la quale controlla la ricezione del relativo messaggio di *acknoledgement*, come mostrato in Figura 42. 3. Si compone la procedura di test come mostrato in Figura 43. 4. Infine si clicca sull'icona "*run procedure*" e si associa

il file "*SocketTestTool.jar*", mostrato in Figura 35.

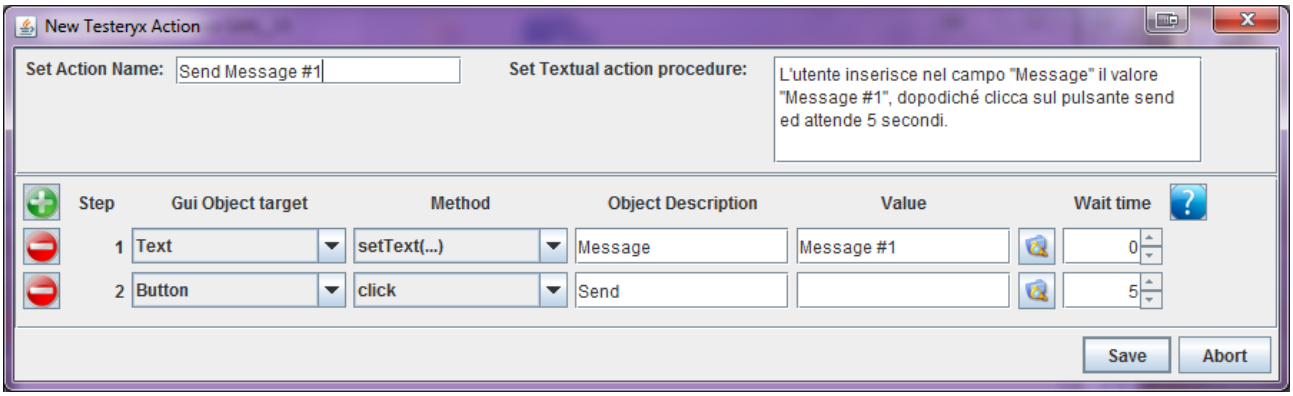

Figura 41 Definizione sequenza di azioni per l'invio del messaggio di tipo 1.

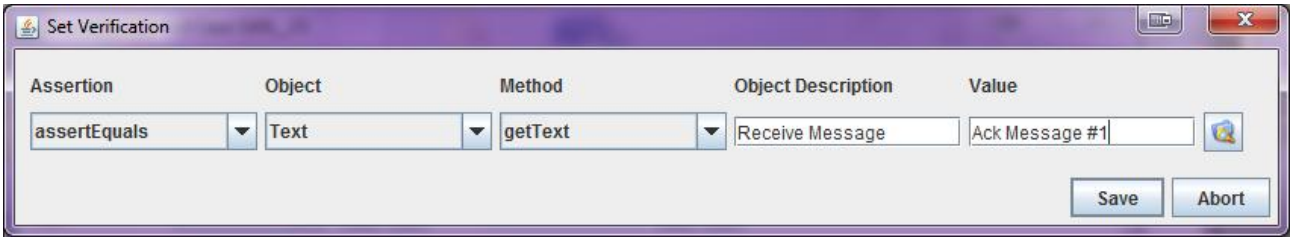

Figura 42 Definizione della verifica di ricezione del relativo *Acknowledgement*.

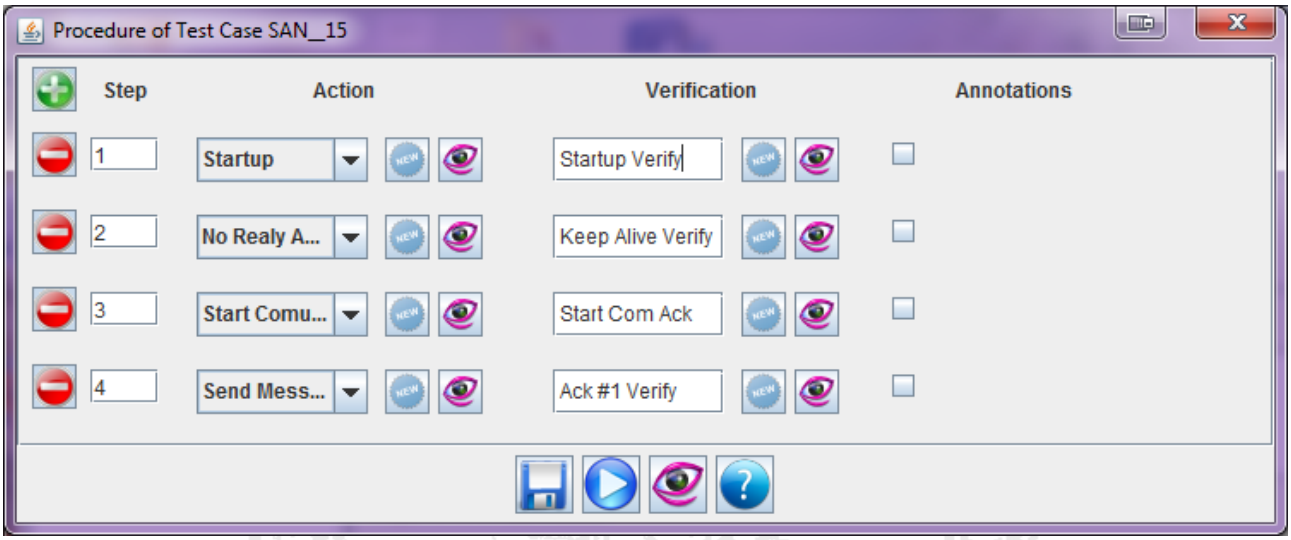

Figura 43 Composizione della procedura di test.

4.3 Valutazione dei casi di test

All'inizio della fase di sperimentazione sono emerse due problematiche rilevanti. Una prima riguarda il fatto che "*Maveryx*" non riesce ad localizzare gli elementi grafici se l'interfaccia utente non è stata progettata secondo precise regole, che agevolano gli algoritmi di ricerca. Ad esempio, si dovrebbero assegnare i *toolTipText* a pulsanti costituiti da solo icone, o a campi testuali che non vengono posizionati alla destra dell'etichette di riferimento. Una seconda problematica è legata al caso reale, il quale ha chiesto la lettura di messaggi contenenti caratteri speciali. Tali caratteri sono stati codificati erroneamente da parte di "*Maveryx*", nel mentre che li digitava all'interno del campo testuale. Per tali ragioni si è dovuto intervenire sull'applicazione "*SocketTest*" (la quale era stata creata *ad-hoc* per testare il CSCI SAN) al fine di fare in modo che tali problematiche venissero eluse.

Una volta risolti tali problemi si è passati all'effettiva esecuzione dei casi di test precedentemente descritti. Al termine dell'esecuzione di ogni test si è preso visione dei relativi file di log. Da ognuno dei quali si è evinto la correttezza del CSCI SAN relativamente ai requisiti coperti. Successivamente, sono state realizzate altre procedure di test, con lo scopo di accertare l'attendibilità dei risultati ottenuti in precedenza, modificando i risultati attesi e/o i messaggi inviati. A tal proposito i file di log ottenuti hanno evidenziato gli errori volontariamente inseriti.

Al termine della definizione dei casi di test, descritti nel paragrafo precedente, sono state apportare alcune modifiche all'applicazione "*SocketTest*". Tali modifiche hanno comportato un riposizionamento degli elementi grafici ed una leggera ridefinizione dei loro attributi visivi. Ciò nonostante, avviando la campagna di test precedentemente definita, si è osservato che gli *script* generati in precedenza si comportavano allo stesso modo. Nella sottostante Figura 44 è mostrato in che modo sia possibile effettuare il *testing* di regressione. Si seleziona il test *suite*

"SAN", dopodiché cliccando su "*run*" viene visualizzata la schermata in cui si selezionano, e successivamente si avviano, i casi di test desiderati.

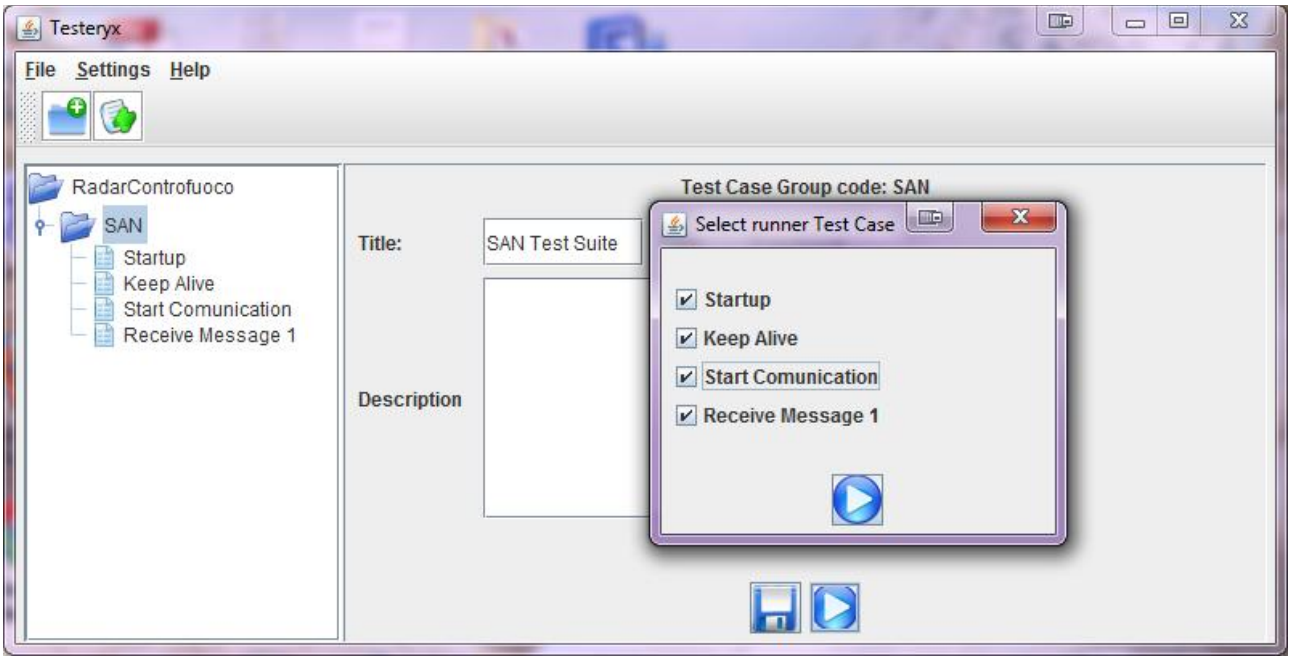

Figura 44 Schermata per l'avvio di un'intera campagna di test.

4.4 Limitazioni

Alla luce dell'attività di sperimentazione svolta, si è giunti al maturare le seguenti considerazioni.

Testeryx utilizza per la generazione degli *script* principalmente le librerie offerte dal *tool Maveryx*. Quest'ultimo, non essendo ancora perfezionato, comporta delle problematiche che si rivesano direttamente su *Testeryx*. La soluzione a riguardo, sarebbe quella di ricercare nuove soluzioni e tecniche per raggirare tali problematiche. Inoltre, ci sarebbe la possibilità di importare nuove tecnologie che in maniera sinergica lavorano tra di loro.

Un'altra limitazione, tipica dei *tool* automatici, è determinata dal fatto che non tutte le possibili esigenze nell'esecuzione delle procedure di test possono essere soddisfatte. A tal proposito, il processo di generazione crea un apposito file Java, che in un secondo momento, permette agli utenti più esperti, di apportare le modifiche desiderate. Tale limitazione col tempo verrà meno, man mano che il *tool*

acquisterà esperienza.

Conclusioni e Sviluppi Futuri

Lo studio effettuato nelle prime fasi del lavoro di tesi ha permesso la realizzazione del *tool* **Testeryx**. Tale applicazione, attraverso l'interfaccia grafica messa a disposizione, consente all'utente di utilizzarla in maniera intuitiva. In questo modo possono essere progettati i casi di test desiderati e generare automaticamente i relativi *script*. In questo modo è possibile collaudare le applicazioni desiderate interagendo solamente con l'interfaccia grafica. Un'altra funzionalità offerta dal sistema progettato e implementato è quello di eseguire gli *script* generati al termine dei quali viene prodotto il relativo *report*, è possibile avviare, anche, l'esecuzione automatica di intere campagne di test. Infine, il tutto è stato predisposto a poter generare, sempre in automatico, la documentazione a corredo delle attività svolte, poichè tutte le informazioni fornite dall'utente e ricavate dal *tool* vengono memorizzate su appositi file XML. In tal modo, è possibile in un secondo momento, estrapolare tali informazioni e generare i relativi STD e STR. Tali documenti possono essere stilati secondo opportune regole di formattazione, come ad esempio, quelle definite dallo standard MIL-STD-498.

Dalle analisi tratte dall'attività di sperimentazione, il *tool* **Testeryx** attualmente realizzato, presenta alcune problematiche, derivanti dal fatto che per la generazione degli *script* si utilizzano principalmente le funzionalità offerte dal *tool* *Maveryx*, unico nel suo genere ma attualmente non è ancora maturo.

Gli sviluppi futuri dovranno interessare diversi aspetti. Il primo aspetto deve riguardare la ricerca e/o lo sviluppo di nuove soluzioni per migliorare l'efficacia dell'automatizzazione; il secondo deve fornire al *tool* la funzionalità di anteprima

ed esportazione documentale delle attività di test svolte; ed infine, in terzo deve fare in modo che sia possibile l'inclusione, anche parziale, di casi di test già implementati per altri scenari.

Appendice A

Codice Implementato

L'appendice A ha lo scopo di mostrare il codice implementato per la realizzazione del *tool Testeryx*. Il linguaggio di programmazione scelto per l'implementazione del *tool* è Java.

Il software realizzato è fortemente ispirato al modello introdotto nel capitolo 3. Come già specificato nel suddetto capitolo, il *pattern* architetturale utilizzato è stato quello dell'MVC. Di seguito, verranno mostrate le implementazioni delle classi relative alla *business logic*, alla parte di controllo, alcune classi riguardanti la *presentation* e alcune classi di utilità.

A.1 Business Logic

Testeryx Test Case Project Class

Questa classe è la principale di tutta la *business logic*, difatti il *controller* ha come unico riferimento l'istanza di tale classe, mediante cui invoca tutte le funzionalità offerte dalla *business logic*.

```
/**********************************************************************
 * Module:  TestCaseProject.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/

package model.TestModel;

import java.io.File;
import java.util.*;
import utility.XMLManager;

public class TestCaseProject {
	private String title;
	private String description;
	private String url;
	private ArrayList<TestCaseGroup> testCaseGroupCollection;
	private XMLManager xml;

	public TestCaseProject(String title, String description, String url) {
		this.title = title;
		this.description = description;
		this.url = url;
		this.xml = new XMLManager(this.url);

		this.testCaseGroupCollection = new ArrayList<TestCaseGroup>();
	}

	@SuppressWarnings("unchecked")
	public void addTestCaseGroup(TestCaseGroup testCaseGroup) {
		new File(this.url + File.separator + testCaseGroup.getCode()).mkdir();

		this.xml.addTestCaseGroup(testCaseGroup.getCode(),
				testCaseGroup.getTitle(), testCaseGroup.getInfo(),
				testCaseGroup.getType());

		this.testCaseGroupCollection.add(testCaseGroup);

		Collections.sort(testCaseGroupCollection, new Comparator() {
			public int compare(Object o1, Object o2) {
				TestCaseGroup p1 = (TestCaseGroup) o1;
				TestCaseGroup p2 = (TestCaseGroup) o2;
				return p1.getCode().compareToIgnoreCase(p2.getCode());
			}
		});
	}

	@SuppressWarnings("unchecked")
	public void setLoadedTestCaseGroup(TestCaseGroup testCaseGroup) {
		this.testCaseGroupCollection.add(testCaseGroup);

		Collections.sort(testCaseGroupCollection, new Comparator() {
			public int compare(Object o1, Object o2) {
				TestCaseGroup p1 = (TestCaseGroup) o1;
				TestCaseGroup p2 = (TestCaseGroup) o2;
				return p1.getCode().compareToIgnoreCase(p2.getCode());
			}

		});
```

```
	}

	public TestCaseGroup getTestCaseGroup(String code) {
		Iterator<TestCaseGroup> iter = testCaseGroupCollection.iterator();
		TestCaseGroup tcg;
		while (iter.hasNext()) {
			tcg = iter.next();
			if (tcg.getCode().equals(code)) {
				return tcg;
			}
		}
		return null;
	}

	public void removeTestCaseGroup(TestCaseGroup testCaseGroup) {
		Iterator<TestCaseGroup> iter = testCaseGroupCollection.iterator();
		while (iter.hasNext()) {
			if (iter.next().getCode() == testCaseGroup.getCode()) {
				this.testCaseGroupCollection.remove(testCaseGroup);
				break;
			}
		}
	}

	public void createXMLProject() {
		new File(this.url).mkdir();
		xml.newTestCaseProcject(title, description);
		xml.createXMLActionList();
	}

	public String getURL() {
		return this.url;
	}

	public String getTitle() {
		return this.title;
	}

}
```

Testeryx Test Case Group Class

Questa classe ha il compito tenere i riferimenti dei casi test ad essa associata.

```
package model.TestModel;

/**********************************************************************
 * Module:  TestCaseProject.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import utility.XMLManager;

public class TestCaseGroup {
	private String code;
```

```
private String title;
private String info;
private String type;
private ArrayList<TestCase> testCaseList;
public TestCaseProject testCaseProjectParent;

public TestCaseGroup(String code, String title, String info, String type,
            TestCaseProject TCProject) {
      this.code = code;
      this.title = title;
      this.info = info;
      this.type = type;
      this.testCaseList = new ArrayList<TestCase>();
      this.testCaseProjectParent = TCProject;
}

public void addTestCase(TestCase testCase) {
      new File(testCaseProjectParent.getURL() + "\\" + this.code + "\\"
                  + testCase.getCodeNumber()).mkdir();
      XMLManager xml = new XMLManager(testCaseProjectParent.getURL());
      xml.addTestCase(testCase.getCodeNumber(), testCase.getName(),
                  testCase.getInfo(), testCase.getRequirement(),
                  testCase.getPrerequisite(), testCase.getTestInputs(),
                  testCase.getResult(), testCase.getCriteria(),
                  testCase.getAssumptions(), this.code);
      this.testCaseList.add(testCase);
}

public void addLoadedTestCase(TestCase testCase) {
      this.testCaseList.add(testCase);

}

public TestCase getTestCase(String code) {
      Iterator<TestCase> iter = testCaseList.iterator();
      TestCase tc;
      while (iter.hasNext()) {
            tc = iter.next();
            if (tc.getCodeNumber().equals(code)) {
                  return tc;
            }
      }
      return null;
}

public void removeTestCase(TestCase testCase) {
      // TODO: implement
}

public String getCode() {
      return this.code;
}

public String getTitle() {
      return this.title;
}

public String getInfo() {
      return this.info;
```

```
	}

	public String getType() {
		return this.type;
	}

}
```

TestCase

L'obiettivo di questa classe è di mantenere tutte le informazioni fornite dall'utente, tra cui la procedura di test.

```
/**********************************************************************
 * Module:  TestCase.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/

import java.util.*;

public class TestCase {
	private String codeNumber;
	private String name;
	private String info;
	private String requirementAddressed;
	private String prerequisite;
	private String testInputs;
	private String result;
	private String criteriaForEvaluatingResults;
	private String assumptioinsConstraints;
	private Procedure procedure;

	public TestCase(String codeNumber, String name, String info,
			String requirement, String prerequisite, String testInputs,
			String result, String criteria, String assumptions) {
		this.codeNumber = codeNumber;
		this.name = name;
		this.info = info;
		this.requirementAddressed = requirement;
		this.prerequisite = prerequisite;
		this.testInputs = testInputs;
		this.result = result;
		this.criteriaForEvaluatingResults = criteria;
		this.assumptioinsConstraints = assumptions;
	}

	public void setProcedure(String url, String codeTestCase) {
		this.procedure = new Procedure(url, codeTestCase);
	}

	public void createXMLProcedure() {
		this.procedure.createXMLProcedure();
	}

	public Procedure getProcedure() {
		return this.procedure;
```

```
	}

	public boolean hasProcedure() {

		if (this.procedure == null) {
			return false;
		}
		return true;
	}

	public String getCodeNumber() {
		return this.codeNumber;
	}

	public String getName() {
		return this.name;
	}

	public String getInfo() {
		return this.info;
	}

	public String getRequirement() {
		return this.requirementAddressed;
	}

	public String getResult() {
		return this.result;
	}

	public String getTestInputs() {
		return this.testInputs;
	}

	public String getCriteria() {
		return this.criteriaForEvaluatingResults;
	}

	public String getAssumptions() {
		return this.assumptioinsConstraints;
	}

	public String getPrerequisite() {
		return this.prerequisite;
	}

}
```

Procedure

La classe *procedure* permette la gestione dei singoli passi di una procedura di test.

```
/**********************************************************************
 * Module:  Procedure.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/
package model.TestModel;
import java.util.*;

import utility.XMLManager;

public class Procedure {
	private Integer stepCaseCount;
	private ArrayList<StepCase> stepCaseList;
	private String codeTestCase;
	private String url;
	private XMLManager xml;

	public Procedure(String url, String codeTestCase) {
		this.stepCaseCount = 0;
		this.stepCaseList = new ArrayList<StepCase>();
		this.codeTestCase = codeTestCase;
		this.url = url;
		this.xml = new XMLManager(url);
	}

	public void createXMLProcedure() {
		this.xml.createXMLProcedure(codeTestCase);
	}

	public void addStepCase(String stepCaseCount, MacroAction macroAction,
			Verification verification, String annotation) {
		StepCase sc = new StepCase(stepCaseCount, macroAction, verification,
				annotation);
		this.stepCaseList.add(sc);
	}

	public void saveProcedure() {
		XMLManager xml = new XMLManager(this.url);
		Iterator<StepCase> iter = this.stepCaseList.iterator();
		StepCase sC = null;
		while (iter.hasNext()) {
			sC = iter.next();
			xml.updateProcedure(this.codeTestCase, sC.getStepCaseNumber(),
					sC.getMacroActionName(), sC.getVerification(),
					sC.getAnnotation());
		}
	}

	public void removeStepCase(StepCase stepCase) {
		// TODO: implement
	}

	public void addStepCaseList(Procedure procedure, int begin, int end) {
		// TODO: implement
	}

}
```

StepCase

Ogni passo di una procedura è composto da una sequenza di azioni, una verifica e da un'annotazione. Pertanto tale classe mantiene traccia di tale composizione.

```
/**********************************************************************
 * Module:  StepCase.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/
package model.TestModel;

import java.util.*;

public class StepCase {
	private int stepCaseNumber;
	private String annotation;
	public MacroAction macroAction;
	public Verification verification;

	public StepCase(String stepCaseNumber, MacroAction macroAction,
			Verification verification, String annotation) {
		this.stepCaseNumber = Integer.parseInt(stepCaseNumber);
		this.macroAction = macroAction;
		this.verification = verification;
		this.annotation = annotation;
	}

	public String getStepCaseNumber() {
		return String.valueOf(this.stepCaseNumber);
	}

	public String getMacroActionName() {
		return this.macroAction.getName();
	}

	public String[] getVerification() {
		String[] verification = { this.verification.getAssertion(),
				this.verification.getGuiObject(),
				this.verification.getMethod(),
				this.verification.getObjDesc(),
				this.verification.getValue(),
				this.verification.getTxt() };
		return verification;
	}

	public String getAnnotation() {
		return this.annotation;
	}


}
```

MacroAction

Si definisce macro azione una sequenza di azioni atomiche eseguire su un'interfaccia grafica in maniera sequenziale. Tale classe mantiene traccia di tale sequenza di azioni atomiche.

```
/**************************************************************************
 * Module:  MacroAction.java
 * Author:  Maurizio Sorrentino
 *************************************************************************/
package model.TestModel;

import java.util.*;
import utility.XMLManager;

public class MacroAction {
	private String actionName;
	private String txt;
	private ArrayList<Action> actionList;

	public MacroAction(String actionName, String txt) {
		this.actionName = actionName;
		this.txt = txt;
		this.actionList = new ArrayList<Action>();
	}

	public void addAction(Action newAction) {
		this.actionList.add(newAction);
	}

	public void saveMacroAction(String url) {
		XMLManager xml = new XMLManager(url);
		xml.addMacroAction(this);
	}

	public String getName() {
		return this.actionName;
	}

	public String getTxt() {
		return this.txt;
	}

	public ArrayList<Action> getActionList() {
		return this.actionList;
	}
```

Action

L'oggetto azione permette di capire cosa l'utente intende fare ad ogni passo.

```
/**************************************************************************
 * Module:  Action.java
 * Author:  Maurizio Sorrentino
 *************************************************************************/
package model.TestModel;

import java.util.*;

public class Action {
	private String method;
	private String guiObject;
	private String guiDescription;
	private String value;
	private String waitConstraint;

	public Action(String method, String guiObject, String guiDescription,
			String value, String waitConstraint) {
		this.method = method;
		this.guiObject = guiObject;
		this.guiDescription = guiDescription;
		this.value = value;
		this.waitConstraint = waitConstraint;
	}

	public String getMethos() {
		return this.method;
	}

	public String getGuiObject() {
		return this.guiObject;
	}

	public String getGuiDescription() {
		return this.guiDescription;
	}

	public String getValue() {
		return this.value;
	}

	public String getWaitConstraint() {
		return this.waitConstraint;
	}

}
```

Verification

Ad ogni macro azione, ovvero ad ogni sequenza di azioni, si definisce una verifica da effettuare sull'interfaccia grafica relativa ad un risultato atteso.

```java
/**************************************************************************
 * Module:  Verification.java
 * Author:  Maurizio Sorrentino
 *************************************************************************/
package model.TestModel;

import java.util.*;

public class Verification {

	private String assertion;
	private String guiObject;
	private String method;
	private String objDescription;
	private String value;
	private String txt;

	public Verification(String assertion, String guiObject, String method,
			String objDescription, String value, String txt) {
		this.assertion = assertion;
		this.guiObject = guiObject;
		this.method = method;
		this.objDescription = objDescription;
		this.value = value;
		this.txt = txt;
	}

	public String getAssertion() {
		return this.assertion;
	}

	public String getGuiObject() {
		return this.guiObject;
	}

	public String getMethod() {
		return this.method;
	}

	public String getObjDesc() {
		return this.objDescription;
	}

	public String getValue() {
		return this.value;
	}

	public String getTxt() {
		return this.txt;
	}
}
```

A2. Controller

La classe *Controller* è colei che riceve tutti gli eventi generati dall'utente, e provvede ad attuare le relative operazioni richieste.

```
/**********************************************************************
 * Module:  Controller.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/

package controller;

import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
import org.w3c.dom.Element;
import utility.MaveryxXML;
import utility.RunUtility;
import utility.XMLManager;
import utility.XmlProcedureToJava;
import model.TestModel.*;
import model.*;

public class TesteryxController {
		private TestCaseProject testCaseProject;
		private ArrayList<MacroAction> macroActionList;
		private String url;


		public TesteryxController(String url){
			this.url=url;
		}

	public void newTestCaseProject(String title, String description) {
		this.testCaseProject= new TestCaseProject(title, description,this.url);
		this.testCaseProject.createXMLProject();
		this.macroActionList = new ArrayList<MacroAction>();

		}

	public void newTestCaseGroup(String code, String title, String info, String
type) {
		TestCaseGroup testCaseGroup= new TestCaseGroup(code,  title,  info,
type, this.testCaseProject);
		testCaseProject.addTestCaseGroup(testCaseGroup);
	}

	public void newTestCase(String codeNumber, String name, String info,
			String requirement, String prerequisite, String testInputs,
			String result, String criteria, String assumptions, String co-
deTCG) {
		TestCaseGroup tcg = this.testCaseProject.getTestCaseGroup(codeTCG);
		if (tcg!=null){
			TestCase testCase=new TestCase(codeNumber, name, info, require-
ment, prerequisite, testInputs, result, criteria, assumptions);
		tcg.addTestCase(testCase);
		}else
```

```
                        System.out.println("not Found");
                }

        public void newProcedure(String testCaseCode) {
                String[] splitString= testCaseCode.split("__");
                TestCaseGroup tcg =
this.testCaseProject.getTestCaseGroup(splitString[0]);
                TestCase tc = tcg.getTestCase(splitString[1]);

tc.setProcedure(this.url+splitString[0]+System.getProperty("file.separator")+
                                splitString[1], testCaseCode);
                tc.createXMLProcedure();
        }

        public void setProcedure(String testCaseCode){
                String[] splitString= testCaseCode.split("__");
                TestCaseGroup tcg =
this.testCaseProject.getTestCaseGroup(splitString[0]);
                TestCase tc = tcg.getTestCase(splitString[1]);

tc.setProcedure(this.url+splitString[0]+System.getProperty("file.separator")+
                                splitString[1], testCaseCode);
        }

        public void addStepCase(String stepCaseNumber, String macroAction, String as-
sertVer, String objVer, String methVer,
                        String objDesc, String valueVer, String txtVer, String annota-
tion,  String testCaseCode) {
                String[] splitString= testCaseCode.split("__");
                TestCaseGroup tcg =
this.testCaseProject.getTestCaseGroup(splitString[0]);
                TestCase tc = tcg.getTestCase(splitString[1]);
                Verification verification = new Verification (assertVer, objVer, meth-
Ver,objDesc, valueVer, txtVer);
                Iterator<MacroAction> iter = macroActionList.iterator();
                MacroAction mAction = null;
                while (iter.hasNext()){
                        mAction = iter.next();
                        if (mAction.getName().equals(macroAction)){break;}
                }
                if (mAction.getName().equals(macroAction)){
                tc.getProcedure().addStepCase(stepCaseNumber, mAction, verification,
annotation); }
        }

        public void updateProcedure(String testCaseCode){
                        String[] splitString= testCaseCode.split("__");
                        TestCaseGroup tcg =
this.testCaseProject.getTestCaseGroup(splitString[0]);
                        TestCase tc = tcg.getTestCase(splitString[1]);
                        tc.getProcedure().saveProcedure();
        }

        public void createMacroAction(String actionName, String txt){
                Iterator<MacroAction> iter = this.macroActionList.iterator();
                MacroAction mAction=null;
                while(iter.hasNext()){
                        mAction=iter.next();
                            if(mAction.getName().equals(actionName)){
```

```
                                System.out.println("Action Name arleady exist ");
                                return ;
                            }
                        }
                    this.macroActionList.add(new MacroAction(actionName, txt));
            }

            public void addAction(String actionName, String method, String guiObject,
                    String guiDescription, String value, String waitConstraint){
                Iterator<MacroAction> iter = this.macroActionList.iterator();
                MacroAction mAction=null;
                while(iter.hasNext()){
                    mAction=iter.next();
                        if(mAction.getName().equals(actionName)){
                            break;
                        }
                    mAction=null;
                }
                Action action = new Action(method, guiObject, guiDescription, value,
waitConstraint);
                if(mAction!=null){
                     mAction.addAction(action);
                }else
                    System.out.println("MacroAction not Found!!!");

            }

            public void saveMacroAction(String actionName){
                Iterator<MacroAction> iter = this.macroActionList.iterator();
                MacroAction mAction=null;
                while(iter.hasNext()){
                    mAction=iter.next();
                        if(mAction.getName().equals(actionName)){
                            break;
                        }
                    mAction=null;
                }
                if(mAction!=null){
                        mAction.saveMacroAction(this.url);
                }else
                        System.out.println("MacroAction not Found!!!");

            }

            public void setPreferences(String jrePath, String maveryxPath)       {
               Preferences preferences= new Preferences();
               preferences.setPreference(jrePath, maveryxPath);
            }

            public String getUrlProject(){
                return this.url;
            }

          public void procedure(String testCaseCode) {
                 String[] splitString= testCaseCode.split("__");
                   TestCaseGroup tcg =
this.testCaseProject.getTestCaseGroup(splitString[0]);
                TestCase tc = tcg.getTestCase(splitString[1]);
                if(!tc.hasProcedure()){ //if not exist
```

```
                        newProcedure(testCaseCode);
                }

        }

        public void runProcedure(String testCaseCode) {
                XmlProcedureToJava xmlProcToJava = new XmlProcedureToJava(testCaseCode,
this.url);
                xmlProcToJava.loadProcedureXML();
                xmlProcToJava.createProcedureJavaFile();
                RunUtility run = new RunUtility(this.url);
                run.runProcedure(testCaseCode);
        }

        public void runOnlyProcedure(String testCaseCode) {
                RunUtility run = new RunUtility(this.url);
                run.runProcedure(testCaseCode);
        }

        public void createMaveryxXML(String appName, String mainClass, String jarPath,
                        String workdir, String vMArg, String mainArg , String testCaseCode)
{
                Preferences preferences= new Preferences();
                MaveryxXML maveryxXML = new MaveryxXML(this.url);
                maveryxXML.createMaveryxXML(appName, mainClass, jarPath, workdir, vMArg,
mainArg,
                                preferences.getJrePath(), preferences.getMaveryxPath(),
testCaseCode);
        }


        public void loadTestCaseProject() {
                 XMLManager xml = new XMLManager(this.url);
                 String[] project = xml.loadTestCaseProject();
                 this.macroActionList = new ArrayList<MacroAction>();

                 this.testCaseProject= new TestCaseProject(project[0],
project[1],this.url);
                 for (int i=2; i<project.length; i++){
                        String[] tCGInfoArray = xml.getTCGInfo(project[i]);
                        TestCaseGroup tCG = new TestCase-
Group(tCGInfoArray[0],tCGInfoArray[1],
                                        tCGInfoAr-
ray[2],tCGInfoArray[3],this.testCaseProject);
                        String[] tCaseList = xml.getTCCodeList(project[i]);
                        for (int j=0; j<tCaseList.length;j++){
                                String [] tCaseInfo = xml.getTCInfo(project[i], tCaseL-
ist[j]);
                                TestCase tCase = new Test-
Case(tCaseInfo[0].split("__")[1],tCaseInfo[1],tCaseInfo[2],tCaseInfo[3],
                                                tCaseIn-
fo[4],tCaseInfo[5],tCaseInfo[6],tCaseInfo[7],tCaseInfo[8]);
                                tCG.addLoadedTestCase(tCase);
                        }
                        this.testCaseProject.setLoadedTestCaseGroup(tCG);
                 }
          }
}
```

A3. Viewer

Il *viewer* si occupa della gestione di tutti servizi che la *presentation logic* offre al servizio. Tutte le altre classi che si occupano dell'interfaccia grafica, fanno capo a tale classe per invocare le relative richieste.

```
/**************************************************************************
 * Module:  TesteryxGuiFrame.java
 * Author:  Maurizio Sorrentino
 *************************************************************************/
package view;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.Toolkit;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.net.URL;
import java.util.ArrayList;
import java.util.Iterator;
import javax.swing.ComboBoxModel;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JSeparator;
import javax.swing.border.EmptyBorder;
import utility.XMLManager;
import controller.TesteryxController;
import view.dialog.*;
import view.panel.*;
import view.other.*;
import java.awt.Color;
import javax.swing.UIManager;

public class TesteryxGuiFrame extends JFrame {

	private JPanel contentPane;

	private ClassLoader cl = getClass().getClassLoader();
	private ImageIcon icnNewProject=new ImageIcon(
cl.getResource("images/newProject.png"));
	private ImageIcon icnLoadProject=new ImageIcon(
cl.getResource("images/openProject.png"));
	private ImageIcon icnPref= new ImageI-
con(cl.getResource("images/preferences.jpg"));
```

```
	private ImageIcon addStepIcon=new ImageI-
con(cl.getResource("images/addStep.jpg"));
	private ImageIcon helpIcon= new ImageIcon(cl.getResource("images/help.png"));
	private ImageIcon removeIcon= new ImageI-
con(cl.getResource("images/deleteStep.png"));
	private ImageIcon urlBrowse= new ImageIcon(cl.getResource("images/browse.png"));
	private ImageIcon leafTreeIcon=  new ImageI-
con(cl.getResource("images/JTreeLeaf.jpg"));
	private ImageIcon openTreeIcon=  new ImageIcon(
cl.getResource("images/JTreeOpen.png"));
	private ImageIcon closeTreeIcon=  new ImageI-
con(cl.getResource("images/JTreeClose.png"));
	private ImageIcon addTestGroupIcon= new ImageI-
con(cl.getResource("images/addTCG.png"));
	private ImageIcon addTestCaseIcon = new ImageI-
con(cl.getResource("images/addTC.png"));
	private ImageIcon saveIcon= new ImageIcon( cl.getResource("images/save.png"));
	private ImageIcon runIcon= new ImageIcon(
cl.getResource("images/runProcedure.png"));
	private ImageIcon procedureIcon= new ImageIcon(
cl.getResource("images/procedure.png"));
	private ImageIcon newIcon= new ImageIcon(cl.getResource("images/new.png"));
	private ImageIcon viewIcon= new ImageIcon(cl.getResource("images/view.png"));
	private ImageIcon preViewIcon= new ImageI-
con(cl.getResource("images/preview.png"));



	private TesteryxController testeryxController;
	private XMLManager xmlManager;
	private ToolBarTesteryx toolBar;
	private TCProject tCProject;

	private String urlProject;

	private JFileChooser urlChooser;


	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					TesteryxGuiFrame frame = new TesteryxGuiFrame();
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public TesteryxGuiFrame() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 800, 600);
		setTitle("Testeryx");
		contentPane = new JPanel();
```

```
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		contentPane.setLayout(new BorderLayout(0, 0));
		setContentPane(contentPane);

		JMenuBar menuBar = new JMenuBar();
		menuBar.setBackground(UIManager.getColor("List.selectionBackground"));
		setJMenuBar(menuBar);

		JMenu mnFile = new JMenu("File");
		mnFile.setMnemonic('F');
		menuBar.add(mnFile);

		JMenuItem mntmNewTestCase = new JMenuItem("New Project");
		mntmNewTestCase.setMnemonic('N');
		mntmNewTestCase.setToolTipText("create a new Test Case Project");
		mntmNewTestCase.setIcon(icnNewProject);
		mntmNewTestCase.addMouseListener(new MouseAdapter() {
			public void mousePressed(MouseEvent e) {
				if (TesteryxGuiFrame.this.isMaximumSizeSet()) {
				Dimension screen-
Size=Toolkit.getDefaultToolkit().getScreenSize();
				int screenWindth=(int) screenSize.getWidth();
				int screenHeight=(int) screenSize.getHeight();

				float div=(float) 1.1;
				float d=(float) 1.3;
				TesteryxGuiFrame.this.setSize((int) (screenWindth/div), (int)
(screenHeight/d));
				}
				NewTCProjectDialog newTcProject = new NewTCProjectDia-
log(TesteryxGuiFrame.this);
			}

		});
		mnFile.add(mntmNewTestCase);

		JMenuItem mntmLoadTestCase = new JMenuItem("Open Project");
		mntmLoadTestCase.setMnemonic('O');
		mntmLoadTestCase.setToolTipText("open a Test Case Project");
		mntmLoadTestCase.setIcon(icnLoadProject);
		urlChooser=new JFileChooser();
		urlChooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
		mntmLoadTestCase.addMouseListener(new MouseAdapter() {
			public void mousePressed(MouseEvent e) {
				int return-
Val=urlChooser.showOpenDialog(TesteryxGuiFrame.this);

				if (returnVal==JFileChooser.APPROVE_OPTION){
					File file=urlChooser.getSelectedFile();
					loadTestCasePro-
ject(file.getAbsolutePath()+File.separator);
				}
			}
		});
		mnFile.add(mntmLoadTestCase);

		JMenu mnEdit = new JMenu("Settings");
		mnEdit.setMnemonic('s');
		menuBar.add(mnEdit);
```

```
            JMenuItem mntmSetting = new JMenuItem("Preferences");
            mntmSetting.setMnemonic('P');
            mntmSetting.setIcon(icnPref);
            mntmSetting.addMouseListener(new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {
                        Preferences preferences= new Prefe-
rences(TesteryxGuiFrame.this);
                  }
            });
            mnEdit.add(mntmSetting);

            JMenu mnHelp = new JMenu("Help");
            mnHelp.setMnemonic('H');
            menuBar.add(mnHelp);

            JMenuItem mntmAbout = new JMenuItem("About");
            JSeparator bar = new JSeparator();
            mntmAbout.addMouseListener(new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {

                  }
            });

            mnHelp.add(mntmAbout);
            mnHelp.add(bar);

            JMenuItem mntmHelp = new JMenuItem("Help");

            mntmHelp.addMouseListener(new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {

                  }
            });

            mnHelp.add(mntmHelp);

      }

      public void createTestCaseProject(String title, String description, String url){
            this.testeryxController = new TesteryxControl-
ler(url+System.getProperty("file.separator")+title+System.getProperty("file.separator")
);
            this.testeryxController.newTestCaseProject(title, description);
            this.xmlManager = new XMLManag-
er(url+System.getProperty("file.separator")+title+System.getProperty("file.separator"))
;
            this.tCProject = new TCProject(this);
            toolBar= new ToolBarTesteryx(this);
            this.urlProject = url;

            JPanel projectPanel = new JPanel();

            projectPanel.setBorder(new EmptyBorder(0, 0, 0, 0));
            projectPanel.setLayout(new BorderLayout(0, 0));

            projectPanel.add(toolBar, BorderLayout.NORTH);
            projectPanel.add(tCProject, BorderLayout.CENTER);
```

```
            setContentPane(projectPanel);

            this.setVisible(true);

      }

      public void loadTestCaseProject(String url){
            this.testeryxController = new TesteryxController(url);
            this.testeryxController.loadTestCaseProject();
            this.xmlManager = new XMLManager(url);
            this.toolBar= new ToolBarTesteryx(this);
            this.tCProject = new TCProject(this);
            this.urlProject = url;

            JPanel projectPanel = new JPanel();
            projectPanel.setBorder(new EmptyBorder(0, 0, 0, 0));
            projectPanel.setLayout(new BorderLayout(0, 0));

            projectPanel.add(toolBar, BorderLayout.NORTH);
            projectPanel.add(tCProject, BorderLayout.CENTER);

            setContentPane(projectPanel);

            this.setVisible(true);
      }

      public void addTCG(String code, String title, String info, String type) {

            this.testeryxController.newTestCaseGroup(code, title, info, type);
            this.tCProject.updateJTree();

      }

      public void addTC(String number, String name, String info, String requirement,
                  String prereq, String testInputs, String expeted, String criteria,
String assumption, String codeTCG) {
            this.testeryxController.newTestCase(number, name, info, requirement, pre-
req, testInputs,
                        expeted, criteria, assumption, codeTCG);
            this.tCProject.updateJTree();

      }

      public void setToolBar(){
            this.toolBar.setVisible(true);
      }

      public String[] getInfoProject(){
            return this.xmlManager.getProjectInfo();
      }

      public String[] getTCgroupList() {
            return this.xmlManager.getTCgroupList();

      }

      public String[] getTCList(String tcGroupCode) {
            return xmlManager.getTCList(tcGroupCode);
      }
```

```
	public String[] getTCGInfo(String tcGroupCode) {

		return xmlManager.getTCGInfo(tcGroupCode);
	}

	public String[] getTCInfo(String tcgCode, String tcName) {

		return xmlManager.getTCInfo(tcgCode,tcName);
	}

	public void procedure(String codeTestCase) {
		//xmlManager.createXMLProcedure(codeTestCase);
		this.testeryxController.procedure(codeTestCase);
	}

	public String[] getMacroActionList() {
		return xmlManager.getMacroActionList();
	}

	public String getMacroTxt(String macroActionName) {

		return xmlManager.getMacroTxt(macroActionName);
	}

	public void addMacroAction(ArrayList<StepBean> stepBean, String actionName,
String txt) {
		this.testeryxController.createMacroAction(actionName, txt);
		Iterator<StepBean> iter = stepBean.iterator();
		StepBean stBean;
		while(iter.hasNext()){
			stBean = iter.next();
			this.testeryxController.addAction(actionName,
stBean.getMethodCombo().getSelectedItem().toString(),
				stBean.getTargetCombo().getSelectedItem().toString(),
stBean.getDescriptionField().getText(),
				stBean.getValueField().getText(),
stBean.getWaitSpinner().getValue().toString());


		}
		this.testeryxController.saveMacroAction(actionName);


	}

	public void updateProcedure(ArrayList<StepCaseBean> stepCaseList, String testCa-
seCode) {
		Iterator<StepCaseBean> iter = stepCaseList.iterator();
		StepCaseBean stCaseBean;
		this.testeryxController.newProcedure(testCaseCode);
		while(iter.hasNext()){
			stCaseBean = iter.next();

	this.testeryxController.addStepCase(stCaseBean.getStNumberField().getText(),
				stCaseBean.getmActPanel().getMacroActionName(),
				stCase-
Bean.getVerPanel().getAssert(),stCaseBean.getVerPanel().getGuiObj(),
```

```
                        stCaseBean.getVerPanel().getMeth(),    stCase-
Bean.getVerPanel().getDesc(),
                        stCaseBean.getVerPanel().getValue(), stCase-
Bean.getVerPanel().getTxt(),
                        stCaseBean.getAnnPanel().getAnnotationText(), testCaseCode);

            }
            this.testeryxController.updateProcedure(testCaseCode);
      }

      public void runProcedure(String testCaseCode) {
            this.testeryxController.runProcedure(testCaseCode);
      }

      public void createMaveryxXML(String appName, String mainClass,
                  String jarPath, String workdir, String vMArg, String mainArg, String
testCaseCode) {

      this.testeryxController.createMaveryxXML(appName,mainClass,jarPath,workdir,vMArg,
mainArg, testCaseCode);
      }

      public String[] getTCHaveProcedureList(String testCaseGroupCode) {
            String [] listCode = this.xmlManager.getTCCodeList(testCaseGroupCode);
            String [] listName = this.xmlManager.getTCList(testCaseGroupCode);
            //String [] listName = this.xmlManager.getTCList(testCaseGroupCode);
            FileInputStream file = null;
            ArrayList<String> arrayString = new ArrayList<String>();
            for (int i=0; i<listCode.length;i++){
                  try {
                        file = new FileInput-
Stream(this.urlProject+File.separator+testCaseGroupCode+

      File.separator+listCode[i].split("__")[1]+File.separator+listCode[i]+".java");
                        arrayString.add(listName[i]);
                        arrayString.add(listCode[i]);
                        System.out.println("Aggiunto "+listName[i]+"-"+listCode[i]);
                  } catch (FileNotFoundException e) {
                        //e.printStackTrace();
                        System.out.println("Non aggiunto "+listName[i]+"-
"+listCode[i]);

                  }
            }
            String [] result = new String[arrayString.size()];
            for (int j=0; j<arrayString.size();j++){
                  result[j]= arrayString.get(j);
            }
            return  result;
      }

      public ImageIcon getIcnNewProject() {
            return icnNewProject;
      }

      public ImageIcon getIcnPref() {
            return icnPref;
      }
```

```
public ImageIcon getAddStepIcon() {
	return addStepIcon;
}

public ImageIcon getHelpIcon() {
	return helpIcon;
}

public ImageIcon getRemoveIcon() {
	return removeIcon;
}

public ImageIcon getUrlBrowse() {
	return urlBrowse;
}

public ImageIcon getLeafTreeIcon() {
	return leafTreeIcon;
}

public ImageIcon getOpenTreeIcon() {
	return openTreeIcon;
}

public ImageIcon getCloseTreeIcon() {
	return closeTreeIcon;
}

public ImageIcon getAddTestGroupIcon() {
	return addTestGroupIcon;
}

public ImageIcon getAddTestCaseIcon() {
	return addTestCaseIcon;
}

public ImageIcon getSaveIcon() {
	return saveIcon;
}

public ImageIcon getRunIcon() {
	return runIcon;
}

public ImageIcon getProcedureIcon() {
	return procedureIcon;
}

public ImageIcon getNewIcon() {
	return newIcon;
}

public ImageIcon getViewIcon() {
	return viewIcon;
}

public ImageIcon getPreViewIcon() {
	return preViewIcon;
}
```

```
	public void setAddTCEnable() {
		this.toolBar.setTCGEnabled();
	}

	public void runOnlyProcedure(String testCaseCode) {
		this.testeryxController.runOnlyProcedure(testCaseCode);
	}
}
```

A4. Utility

XMLManager

La classe XMLManager permette di tenere traccia delle informazioni fornite dall'utente. Questa classe viene modificata dal *controller* e letta dal *viewer*.

```
/**************************************************************************
 * Module:  XMLManager.java
 * Author:  Maurizio Sorrentino
 *************************************************************************/

package utility;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import model.TestModel.Action;
import model.TestModel.MacroAction;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XMLManager {
	private DocumentBuilderFactory docFactory;
	private DocumentBuilder docBuilder;
	private Document doc;
	private String url;

	public XMLManager(String url){
		this.url=url;
```

```
		this.docFactory= DocumentBuilderFactory.newInstance();
		try {
			this.docBuilder= docFactory.newDocumentBuilder();
		} catch (ParserConfigurationException e) {
			e.printStackTrace();
		}
	}

	public void newTestCaseProcject(String title, String description){
		this.doc = docBuilder.newDocument();
		Element rootElement = doc.createElement("TEST_CASE_PROJECT");
		rootElement.setAttribute("Title", title);
		rootElement.setAttribute("info", description);
		this.doc.appendChild(rootElement);
		updateXML(this.url,"TestCaseProject.xml");
	}

	public void addTestCaseGroup(String code, String title, String info, String
type){
		try {
			this.doc = docBuild-
er.parse(this.url+System.getProperty("file.separator")+"TestCaseProject.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root= doc.getDocumentElement();
		Element group = doc.createElement("TEST_CASE_GROUP");
		group.setAttribute("Code", code);
		group.setAttribute("Title", title);
		group.setAttribute("info", info);
		group.setAttribute("type", type);
		root.appendChild(group);
		updateXML(url,"TestCaseProject.xml");
	}

	public void addTestCase(String codeNumber, String name, String info, String re-
quirement, String prerequisite,String testInputs, String result, String criteria,
String assumptions, String codeGroup){
		try {
			this.doc = docBuild-
er.parse(this.url+System.getProperty("file.separator")+"TestCaseProject.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		NodeList groups= doc.getElementsByTagName("TEST_CASE_GROUP");
		Element group = null;
		for (int i=0; i<groups.getLength(); i++){

	if(groups.item(i).getAttributes().item(0).getNodeValue().equals(codeGroup))
			{
				group = (Element) groups.item(i);

			}
		}
		Element tc = doc.createElement("TEST_CASE");
```

```
            tc.setAttribute("Code",codeGroup+"__"+codeNumber);
            tc.setAttribute("Name", name);
            tc.setAttribute("info", info);
            tc.setAttribute("requirements", requirement);
            tc.setAttribute("prerequisite", prerequisite);
            tc.setAttribute("testInputs", testInputs);
            tc.setAttribute("results", result);
            tc.setAttribute("criteria", criteria);
            tc.setAttribute("assumptions", assumptions);
            group.appendChild(tc);
            updateXML(url,"TestCaseProject.xml");
      }

      public void createXMLActionList(){
            this.doc = docBuilder.newDocument();
            Element rootElement = doc.createElement("ACTION_LIST");
            this.doc.appendChild(rootElement);
            updateXML(this.url,"ActionList.xml");
      }

      public void createXMLProcedure(String codeTestCase){
            this.doc= this.docBuilder.newDocument();
            Element rootElement=doc.createElement("Procedure");
            rootElement.setAttribute("TestCaseRef", codeTestCase);
            this.doc.appendChild(rootElement);
            File oldProcedure = new File(this.url+File.separator+"Procedure.xml");
            oldProcedure.delete();
            updateXML(this.url,"Procedure.xml");
      }

      public void updateProcedure(String codeTestCase, String stepCaseNumber, String
macroActionName,
                  String[] verification, String annotation ){
            try {
                  this.doc=
this.docBuilder.parse(this.url+System.getProperty("file.separator")+"Procedure.xml");
            } catch (SAXException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
            Element root = this.doc.getDocumentElement();
                  Element stepEl = this.doc.createElement("STEP_CASE");
                  stepEl.setAttribute("Number", stepCaseNumber);
                        Element macroActionEl =
this.doc.createElement("MACRO_ACTION");
                              macroActionEl.setAttribute("Name", macroActionName);
                        Element verifEl = this.doc.createElement("VERIFICATION");
                              verifEl.setAttribute("Assertion", verification[0]);
                              verifEl.setAttribute("GuiObject", verification[1]);
                              verifEl.setAttribute("Method", verification[2]);
                              verifEl.setAttribute("ObjDescr", verification[3]);
                              verifEl.setAttribute("Value", verification[4]);
                              verifEl.setAttribute("txt", verification[5]);
                        Element annotEl = this.doc.createElement("ANNOTATION");
                              annotEl.setAttribute("txt", annotation);
                  stepEl.appendChild(macroActionEl);
```

```
                stepEl.appendChild(verifEl);
                stepEl.appendChild(annotEl);
        root.appendChild(stepEl);
        updateXML(this.url,"Procedure.xml");
}

public void addMacroAction(MacroAction mAction){
        try {
                this.doc = docBuilder.parse(this.url+"ActionList.xml");
        } catch (SAXException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }
        Element root = doc.getDocumentElement();
        Element mActEl = doc.createElement("ACTION");
        mActEl.setAttribute("Name", mAction.getName());
        mActEl.setAttribute("txt", mAction.getTxt());

        Iterator<Action> actionList = mAction.getActionList().iterator();
        Action act;
        int id=0;
        while(actionList.hasNext()){
                act = actionList.next();
                Element actElem = doc.createElement("STEP_ACTION");
                actElem.setAttribute("Number", ++id +"");
                actElem.setAttribute("method", act.getMethos());
                actElem.setAttribute("object", act.getGuiObject());
                actElem.setAttribute("obj_Desc", act.getGuiDescription());
                actElem.setAttribute("value", act.getValue());
                actElem.setAttribute("wait", act.getWaitConstraint());
                mActEl.appendChild(actElem);
        }
        root.appendChild(mActEl);
        updateXML(this.url, "ActionList.xml");

}

public String[] getProjectInfo(){
        try {
                this.doc= this.docBuilder.parse(this.url+"TestCaseProject.xml");
        } catch (SAXException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }
        Element root = this.doc.getDocumentElement();
        String[] result = {root.getAttribute("Title"), root.getAttribute("info")};
        return result;

}

public String[] getTCgroupList() {
        try {
                this.doc= this.docBuilder.parse(this.url+"TestCaseProject.xml");
        } catch (SAXException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
```

```
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("TEST_CASE_GROUP");
		String[] groupEl = new String[nlist.getLength()];

		for (int i=0; i<nlist.getLength();i++){
			groupEl[i] =
nlist.item(i).getAttributes().getNamedItem("Code").getNodeValue();
		}
		return groupEl;
	}

	public String[] getTCList(String tcGroupCode) {
		try {
			this.doc= this.docBuilder.parse(this.url+"TestCaseProject.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("TEST_CASE_GROUP");
		String[] tcEl = null;
		Element group = null;
		for (int i=0; i<nlist.getLength();i++){
			if ( tcGroup-
Code.equals(nlist.item(i).getAttributes().getNamedItem("Code").getNodeValue())){

				group = (Element) nlist.item(i);
				NodeList tlist = group.getElementsByTagName("TEST_CASE");

				tcEl = new String[tlist.getLength()];
				for (int j=0; j<tlist.getLength(); j++){
					tcEl[j] =
tlist.item(j).getAttributes().getNamedItem("Name").getNodeValue();
				}
				break;
			}
		}
		return tcEl;
	}

	public String[] getTCGInfo(String tcGroupCode) {
		try {
			this.doc= this.docBuilder.parse(this.url+"TestCaseProject.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("TEST_CASE_GROUP");
		String[] tcgInfo = new String[4];

		for (int i=0; i<nlist.getLength();i++){
			if ( tcGroup-
Code.equals(nlist.item(i).getAttributes().getNamedItem("Code").getNodeValue())){

				Element group = (Element) nlist.item(i);
```

```
                        tcgInfo[0] = group.getAttribute("Code");
                        tcgInfo[1] = group.getAttribute("Title");
                        tcgInfo[2] = group.getAttribute("info");
                        tcgInfo[3] = group.getAttribute("type");
                        break;
                    }
                }
                return tcgInfo;
        }

        public String[] getTCInfo(String tcgCode, String tcName) {
                try {
                        this.doc= this.docBuilder.parse(this.url+"TestCaseProject.xml");
                } catch (SAXException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                Element root = this.doc.getDocumentElement();
                NodeList nlist = root.getElementsByTagName("TEST_CASE_GROUP");
                String [] tcInfo = new String [9];
                for (int i=0; i<nlist.getLength();i++){
                        if (
tcgCode.equals(nlist.item(i).getAttributes().getNamedItem("Code").getNodeValue())){

                        Element group = (Element) nlist.item(i);
                        NodeList tlist = group.getElementsByTagName("TEST_CASE");


                        for (int j=0; j<tlist.getLength(); j++){
                                if
(tcName.equals(tlist.item(j).getAttributes().getNamedItem("Name").getNodeValue())
                                                ||
tcName.equals(tlist.item(j).getAttributes().getNamedItem("Code").getNodeValue())){
                                        Element tcase = (Element) tlist.item(j);
                                        tcInfo [0] = tcase.getAttribute("Code");
                                        tcInfo [1] = tcase.getAttribute("Name");
                                        tcInfo [2] = tcase.getAttribute("info");
                                        tcInfo [3] = tcase.getAttribute("requirements");
                                        tcInfo [4] = tcase.getAttribute("prerequisite");
                                        tcInfo [5] = tcase.getAttribute("testInputs");
                                        tcInfo [6] = tcase.getAttribute("results");
                                        tcInfo [7] = tcase.getAttribute("criteria");
                                        tcInfo [8] = tcase.getAttribute("assumptions");
                                }
                        }
                        break;
                    }
                }
                return tcInfo;
        }

        public String[] getMacroActionList() {
                try {
                        this.doc= this.docBuilder.parse(this.url+"ActionList.xml");
                } catch (SAXException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
```

```
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("ACTION");
		String[] groupEl = new String[nlist.getLength()];

		for (int i=0; i<nlist.getLength();i++){
			groupEl[i] =
nlist.item(i).getAttributes().getNamedItem("Name").getNodeValue();
		}
		return groupEl;
	}

	public String getMacroTxt(String macroActionName) {
		try {
			this.doc= this.docBuilder.parse(this.url+"ActionList.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("ACTION");

		for (int i=0; i<nlist.getLength();i++){
	if(nlist.item(i).getAttributes().getNamedItem("Name").getNodeValue().equals(macro
ActionName)){return nlist.item(i).getAttributes().getNamedItem("txt").getNodeValue();
			}
		}
		return null;
	}

	public NodeList getMacroStep(String macroActionName) {
		try {
			this.doc= this.docBuilder.parse(this.url+"ActionList.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("ACTION");

		for (int i=0; i<nlist.getLength();i++){

	if(nlist.item(i).getAttributes().getNamedItem("Name").getNodeValue().equals(macro
ActionName)){
				Element action = (Element)nlist.item(i);

				return action.getElementsByTagName("STEP_ACTION");
			}
		}
		return null;
	}

	public String[] getTCCodeList(String testCaseGroupCode) {
		try {
			this.doc= this.docBuilder.parse(this.url+"TestCaseProject.xml");
		} catch (SAXException e) {
			e.printStackTrace();
```

```
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root = this.doc.getDocumentElement();
		NodeList nlist = root.getElementsByTagName("TEST_CASE_GROUP");
		String[] tcEl = null;
		Element group = null;
		for (int i=0; i<nlist.getLength();i++){
			if ( testCaseGroup-
Code.equals(nlist.item(i).getAttributes().getNamedItem("Code").getNodeValue())){
				group = (Element) nlist.item(i);
				NodeList tlist = group.getElementsByTagName("TEST_CASE");
				tcEl = new String[tlist.getLength()];
				for (int j=0; j<tlist.getLength(); j++){
					tcEl[j] =
tlist.item(j).getAttributes().getNamedItem("Code").getNodeValue();
				}
				break;
			}
		}
		return tcEl;
	}

	public String[] loadTestCaseProject(){
		try {
			this.doc = docBuilder.parse(this.url+"TestCaseProject.xml");
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		Element root= doc.getDocumentElement();
		NodeList list = root.getElementsByTagName("TEST_CASE_GROUP");
		String [] project = new String[list.getLength()+2];
		project[0] = root.getAttribute("Title");
		project[1] = root.getAttribute("info");
		for (int i =2 ; i<project.length;i++){
			project[i] = list.item(i-2).getAttributes().getNamedItem("Code").get
NodeValue();
		}
		return project;
	}

	private void updateXML(String xmlProjectURL, String nameFile){
		TransformerFactory transformerFactory = TransformerFactory.newInstance();
		Transformer transformer=null;
		try {
			transformer = transformerFactory.newTransformer();
			transformer.setOutputProperty(OutputKeys.INDENT, "yes");
	} catch (TransformerConfigurationException e) {
			e.printStackTrace();
		}
		DOMSource source = new DOMSource(this.doc);
		StreamResult result =  new StreamResult(new
File(xmlProjectURL+System.getProperty("file.separator")+nameFile));
		try {
			transformer.transform(source, result);
		} catch (TransformerException e) {
			e.printStackTrace();	} } }
```

XmlProcedureToJava

La classe *XMLProcedureToJava* permette di tradurre la procedura di test definita dall'utente in un file java.

```
/**********************************************************************
 * Module:  XmlProcedureToJava.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/
package utility;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import com.sun.xml.internal.bind.v2.schemagen.xmlschema.List;

public class XmlProcedureToJava {
	private String testCaseCode;
	private String urlProject;
	private String urlTestCase;
	private String maverxJavaPath;
	private Document doc;
	private FileWriter file;

	public XmlProcedureToJava(String testCaseCode, String urlProject){
		this.testCaseCode = testCaseCode;
		this.urlProject = urlProject;
		this.urlTestCase = urlPro-
ject+File.separator+this.testCaseCode.split("__")[0]+

File.separator+this.testCaseCode.split("__")[1]+File.separator;
		this.maverxJavaPath="";
		String [] mJP = this.urlTestCase.split("\\\\");
		for (int i=0; i<mJP.length;i++){
			this.maverxJavaPath = this.maverxJavaPath + mJP[i]+ "\\\\";
		}

	}

	public void loadProcedureXML(){

		DocumentBuilderFactory docFactory = DocumentBuilderFacto-
ry.newInstance();
		DocumentBuilder docBuilder=null;

		try {
```

```
                  docBuilder = docFactory.newDocumentBuilder();

            } catch (ParserConfigurationException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
             try {
                   this.doc = docBuilder.parse(new
File(this.urlTestCase+"Procedure.xml"));
            } catch (SAXException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }

      public void createProcedureJavaFile() {
            try {
                  file=new FileWriter(this.urlTestCase+testCaseCode+".java");
            } catch (IOException e) {
                  e.printStackTrace();
            }
            String importing="import static org.junit.Assert.*;"+
                        "\nimport org.junit.*;"+"\nimport org.maveryx.bootstrap.*;"+
                        "\nimport org.maveryx.core.guiApi.*;"+"\n\npublic class
"+testCaseCode+" ";
            String pathName ="{\n" +
                        "private final String pathName = "+'"'+
this.maverxJavaPath+"Maveryx.xml"+'"' + ";"+"\n" +
"public "+ testCaseCode +"() throws Exception {" +"\n super(); " +"\n}" + "\n"+
"@Before\n"+"public void setUp() throws Exception {\n"
+"Bootstrap.startApplication(pathName);" +"\n}" +"\n"+"@After\n"+"public void tear-
Down() throws Exception {\n" +"Bootstrap.stop(pathName);\n" +"}\n";;
            String test = Test();
            String end = "}\n}\n";
            try {
                  file.write(importing+ pathName + test +end);
            } catch (IOException e) {
                  e.printStackTrace();
            }
            try {
                  file.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }

      private String Test() {
            Element rootElement = this.doc.getDocumentElement();
            NodeList stepCaseList = rootElement.getElementsByTagName("STEP_CASE");
            XMLManager xml = new XMLManager(this.urlProject+File.separator);
            String testCase="@Test\n"+"public void " +this.testCaseCode+"_Test"+ "()
throws Exception{\n";
            String method;String object; String obj_Desc; String value;String wait;
            String assertion; String assObj; String assMeth; String assObjDesc; String
assValue;
            MacroActionScript macroActionScript = new MacroActionScript();
            AssertionScript assertionScript = new AssertionScript();
            for (int i=0; i<stepCaseList.getLength();i++){
```

```
                    NodeList stepAction =
xml.getMacroStep(stepCaseList.item(i).getChildNodes().item(1).getAttributes().getNamedI
tem("Name").getNodeValue());
                    for (int j=0; j<stepAction.getLength(); j++){
object = stepAction.item(j).getAttributes().getNamedItem("object").getNodeValue();
                            method = stepAc-
tion.item(j).getAttributes().getNamedItem("method").getNodeValue();
                            obj_Desc = stepAc-
tion.item(j).getAttributes().getNamedItem("obj_Desc").getNodeValue();
                            value = stepAc-
tion.item(j).getAttributes().getNamedItem("value").getNodeValue();
                            wait  = stepAc-
tion.item(j).getAttributes().getNamedItem("wait").getNodeValue();
                            testCase = testCase + macroAction-
Script.getScript(object,method,obj_Desc,value,wait,i,j);
                    }

                    assertion = stepCaseL-
ist.item(i).getChildNodes().item(3).getAttributes().getNamedItem("Assertion").getNodeVa
lue();
                    assObj = stepCaseL-
ist.item(i).getChildNodes().item(3).getAttributes().getNamedItem("GuiObject").getNodeVa
lue();
                    assMeth = stepCaseL-
ist.item(i).getChildNodes().item(3).getAttributes().getNamedItem("Method").getNodeValue
();
                    assObjDesc = stepCaseL-
ist.item(i).getChildNodes().item(3).getAttributes().getNamedItem("ObjDescr").getNodeVal
ue();
                    assValue = stepCaseL-
ist.item(i).getChildNodes().item(3).getAttributes().getNamedItem("Value").getNodeValue(
);
                    testCase = testCase + assertionScript.getAssScript(assertion, as-
sObj, assMeth, assObjDesc, assValue, i);
                }
            return testCase;
        }


}
```

RunUtility

Dopo aver generato il codice Java relativo ad una procedura di test, per poter avviare l'attività di collaudo viene invocata tale classe di utilità. La *RunUtility* ha il compito di individuare compilare ed eseguire il file java associato alla procedura selezionata dall'utente. Al termine dell'esecuzione della procedura viene generato il file di log.

```
/**********************************************************************
 * Module:  RunUtility.java
 * Author:  Maurizio Sorrentino
 **********************************************************************/
```

```
package utility;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.List;
import java.util.Properties;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class RunUtility {
	private String urlProject;
	private String testCaseCode;
	private String urlTestCase;

	public RunUtility(String urlProject) {
	this.urlProject = urlProject;
	}

	public void runProcedure(String testCaseCod){
		this.testCaseCode = testCaseCod;
		this.urlTestCase = this.urlProject+this.testCaseCode.split("__")[0]+
			File.separator+this.testCaseCode.split("__")[1]+File.separator;
			try{
			Properties prop=new Properties();
			FileInputStream filep = new FileInput-
Stream(".\\Testeryx.properties");
			prop.load(filep);
			String directoryMav = prop.getProperty("Maveryx_path");
			String directoryJRE= prop.getProperty("JRE_path");
			filep.close();
			InputStream in = new FileInput-
Stream(this.urlTestCase+this.testCaseCode+".java");
			OutputStream out = new FileOutput-
Stream(".\\"+this.testCaseCode+".java");
			byte[] buffer= new byte[1024];
			int len;
			while((len=in.read(buffer))>0){
				out.write(buffer,0,len);
			}
			in.close();
			out.close();
			FileWriter file=new FileWriter(".\\"+this.testCaseCode+".bat");
			String setClasspath="SET CLASSPATH="+ directory-
Mav+"lib\\JUnit\\org.hamcrest.core_1.1.0.v20090501071000.jar;" +
				directory-
Mav+"lib\\JUnit\\org.junit_4.8.2.v4_8_2_v20110321-1705\\junit.jar;" +
				directory-
Mav+"bin\\bootstrap.jar;"+directoryMav+"bin\\proxy.jar;"+directoryMav+"bin\\utilis.jar;
" +
```

```
                                directory-
Mav+"bin\\jxl.jar;"+directoryMav+"bin\\jdom.jar;"+directoryMav+"bin\\cajo.jar;" +
                                directory-
Mav+"bin\\finderXMLObject.jar;"+directoryMav+"bin\\factory.jar" +
                                directoryJRE+"lib\\rt.jar"+"\n";

                                        String javac= "javac "+ "
.\\"+this.testCaseCode+".java\n exit\n";
      file.write(setClasspath+javac);
                  file.close();
                  String commands = "cmd /c start "+this.testCaseCode+".bat";
                  Runtime.getRuntime().exec(commands);
                  }catch(IOException e){}

                  boolean notFound = false;
                        int i = 0;
                        Result res = null;
                        while(!notFound){
                            try {
                               res= JUnit-
Core.runClasses(Class.forName(this.testCaseCode));
                                        notFound=true;
                                  } catch (ClassNotFoundException e) {
                                        try {
                                              Thread.sleep(1000);
                                              i++;
                                              if(i==10){
                                                    notFound=true;
                                                    e.printStackTrace();
                                              }
                                        } catch (InterruptedException e1) {
                                              e1.printStackTrace();
                                        }
                                  }
                        }
                        FileWriter reportFile = null;
                        try {
                              reportFile=new FileWri-
ter(this.urlTestCase+this.testCaseCode+".txt");
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                        String report ="";
                  if (!res.wasSuccessful()){
                  List<Failure> fail = res.getFailures();
                  for (int k=0; k<fail.size();k++){
                        Failure f = fail.get(k);
                        report = report + "Failure number: "+k+"\n";
                        report = report + "Failure Header: "+f.getTestHeader()+"\n";
                        report = report + "Failure description: "+
f.getDescription()+"\n";
                        report = report + "Failure message: "+ f.getMessage()+"\n";
                        report = report + "Failure trace: "+f.getTrace()+"\n";
                        report = report + "Failure exception: " + f.getException() +
"\n";
                  }
                  } else{
                        report = report + "All Tests succeded\n ";
                  }
```

```
                        try {
                                reportFile.write(report);
                                reportFile.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }

                        File javaFile= new File(".\\"+this.testCaseCode+".java");
                        javaFile.delete();
                        File batFile= new File(".\\"+this.testCaseCode+".bat");
                        batFile.delete();
                        File classFile=new File(".\\"+this.testCaseCode+".class");
                        classFile.delete();
        }
}
```

Appendice B

Guida utente

L'appendice B ha lo scopo di fornire una guida utente a chi volesse utilizzare la *release* del *tool* Testeryx a corredo della tesi.

B1. Introduzione a Testeryx

Testeryx nasce dall'esigenza di avere a disposizione un tool di supporto alle attività di testing. Esso si pone come obiettivo ultimo quello che a partire dal solo SRS è possibile la stesura di documenti quali STP, STD e STR che rispettino delle ben definite regole di formattazione. Testeryx si presta, anche ad essere utilizzato per l'esecuzione di procedure di verifica e validazione su applicazioni aventi un'interfaccia grafica in Java.

Apertura e Configurazione dell'applicazione

Per avviare Testeryx occorre avviare il file Testeryx.jar o cliccando sull'icona due volte, oppure mediante linea di comando digitando, nella cartella in cui è collocato "*java –jar Testeryx.jar*"

Dopodiché verrà visualizzata la seguente schermata principale.

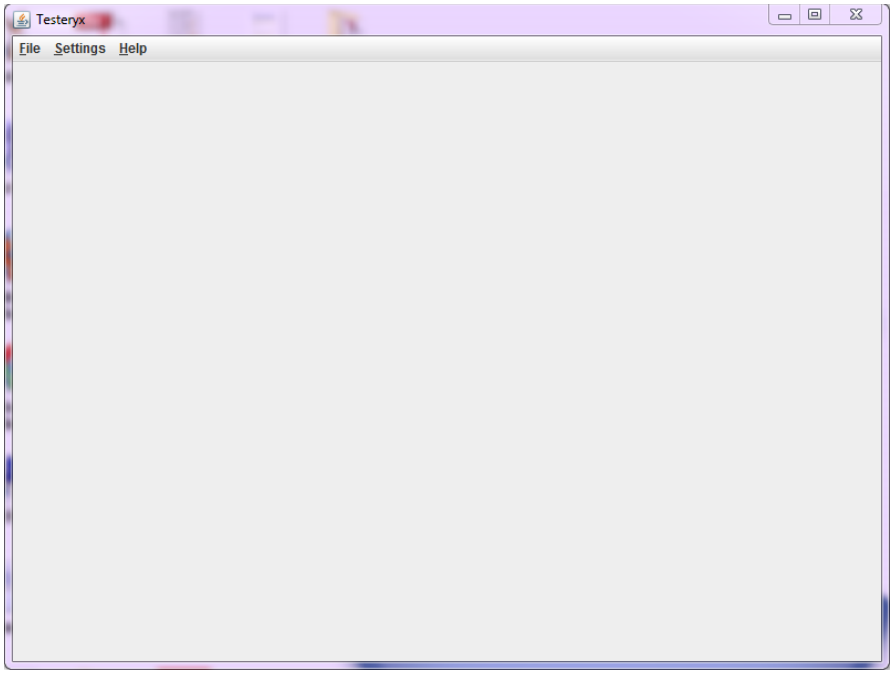

Figura 45 Schermata iniziale.

Come si nota dalla Figura 45, l'interfaccia grafica dell'applicazione è dotata di un menù principale, mediante cui è possibile creare una nuovo "Testeryx Project" oppure caricarne uno (come vedremo più avanti). Se è la prima volta che viene avviato Testeryx è buona norma settare i *path* della *jre* e di *Maveryx* come mostrato in Figura 47. Cliccando su "*Preferences*" viene mostrata una finestra in cui si possono inserire i percorsi.

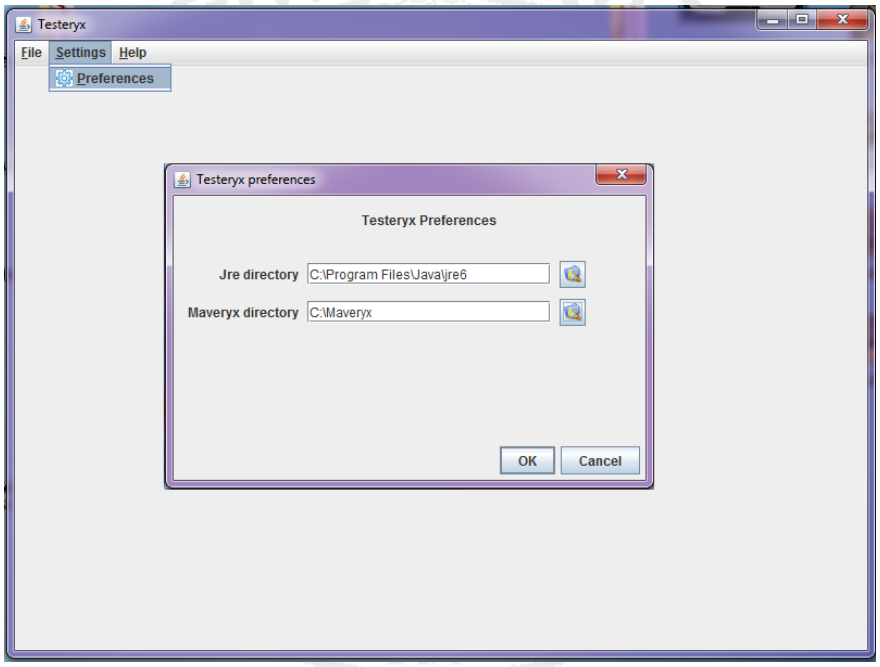

Figura 46 Schermata di configurazione.

B2. Creazione di un Testeryx Project

Per poter creare un nuovo Testeryx Project occorre seguire la seguente procedura, mostrata in Figura 48:

1. dalla Barra del menù clicclare su "new Project";
2. inserire nel campo "Project Name" il nome mneumonico del progetto;
3. inserire nel campo "Description" la descrizione che si vuole dare del progetto; (opzionale)
4. inserire nel campo "Location" il percorso in cui si vuole collocare il progetto;
5. premere sul tasto "OK" per confermare;

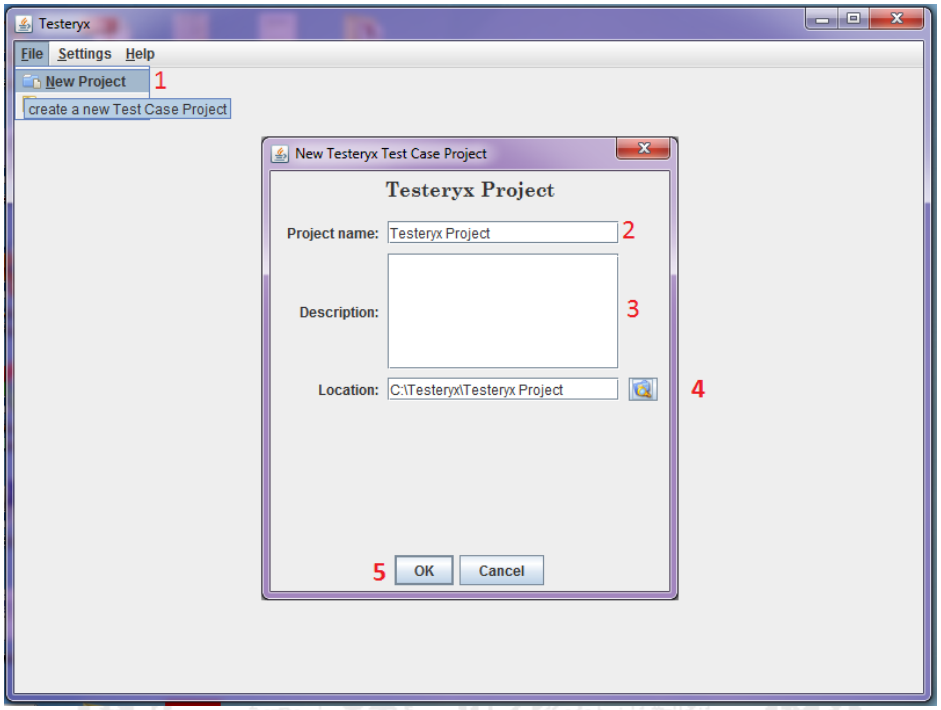

Figura 47 Procedura creazione nuovo progetto.

B3. Aggiunta di un Test Suite

In Figura 49 è mostrata la procedura di aggiunta di un Test Suite al progetto, tale procedura è descritta di seguito:

1. dalla ToolBar cliccare sulla prima icona a partire da sinistra
2. inserire nel campo "Code" il codice da assegnare al Test Suite (obbligato-

rio);

3. inserire nel capo "Title" il titolo associato al Test Suite (opzionale);
4. inserire nel campo "Type" il criterio di aggregazione dei casi di test in esso (opzionale)
5. inserire nel campo "Description" la descrizione del Test Suite.
6. premere il pulsante "OK" per confermare.

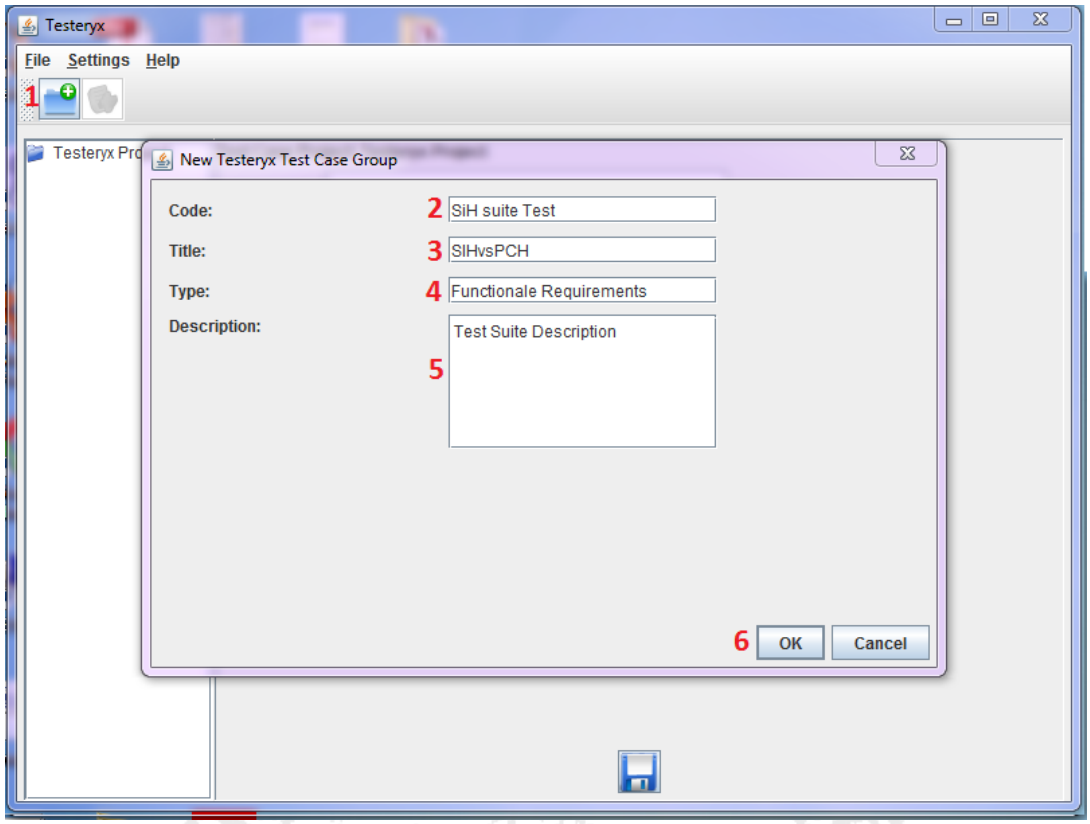

Figura 48 Procedura di aggiunta di un *Test Suite*.

B4. Aggiunta di un Test Case

In Figura 50 viene mostrata la procedura di aggiunta di un Test Case, così come viene dettagliato di seguito:

1. dalla ToolBar cliccare sulla seconda icona a partire dalla sinistra.
2. nella Combo box "Test Case number" occorre definire il numero progressivo del Test Case (obbligatorio);

3. nel campo "Title" occorre inserire il titolo del Test Case (Obbligatorio);
4. nella combo box "Select TCG ref" va selezionato il Test Suite a cui si associa il Test Case (Obbligatorio);
5. dal punto 5 all'11 vi sono una serie di campi in cui viene descritto il Test Case (Opzionali);
6. premere sul pulsante "OK" per confermare.

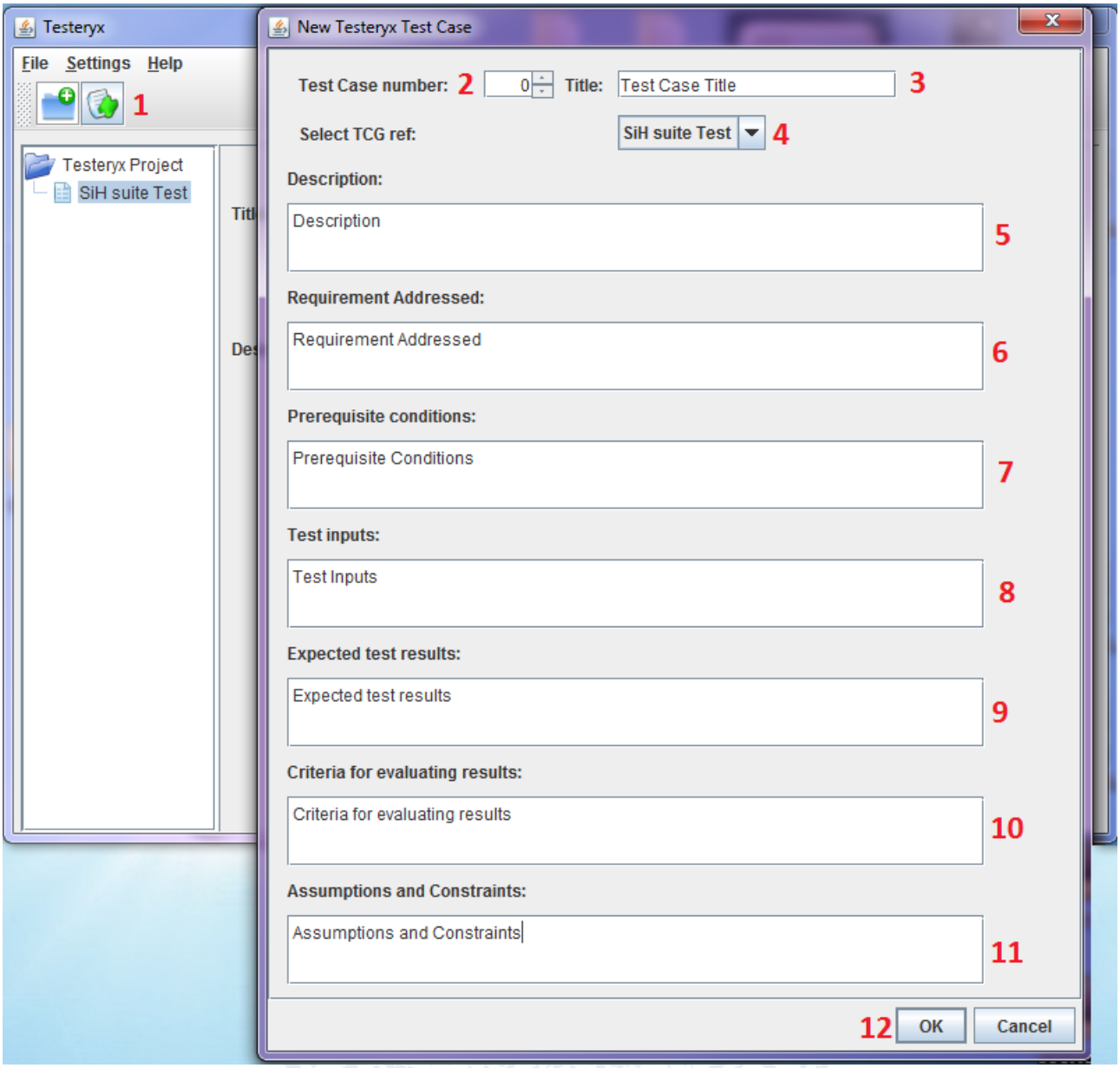

Figura 49 Procedura aggiunta *Test Case*.

B5. Definizione Procedura

Selezionando dalla *Tree View* il Test Case desiderato è possibile assegnare una procedura interattiva nel seguente modo:

1. premere il pulsante recante l'immagine di un libro per aprire la schermata di definizione procedura
2. premere il pulsate di aggiunta step
3. indicare le azioni, le verifiche e le annotazioni da associare ad ogni step (vedi procedura definizione Macro Azioni & Verifiche);
4. premere il pulsante "Salva" per confermare il salvataggio della procedura
5. premere il pulsante di "Run" per avviare la procedura (vedi procedura Esecuzione Test Case)
6. premere il pulsante "View" per avere un'anteprima della documentazione relativa alla procedura.

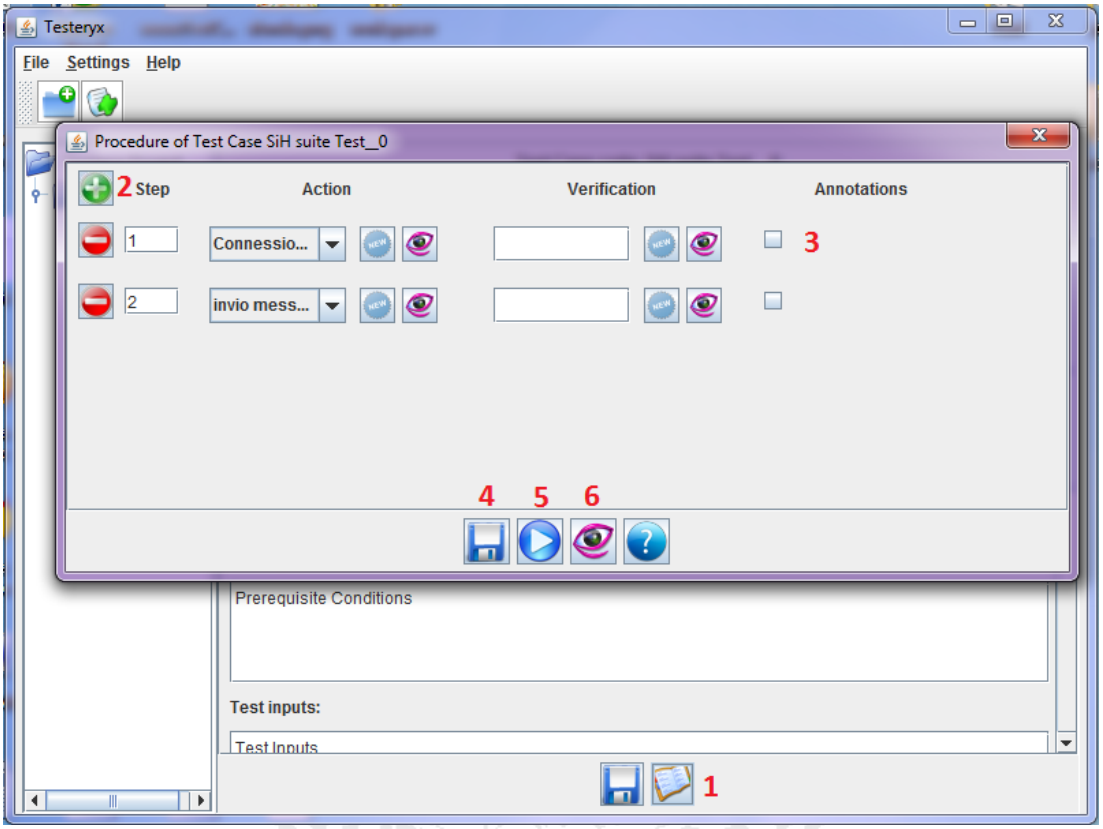

Figura 50 Definizione Procedura *Test Case*.

B5. Definizione Macro Azione e Verifica

Per poter definire una nuova "Macro Azione" occorre:

1. inserire nel campo "Set Action Name" il nome univoco della "Macro Azione";
2. inserire nel campo "Set Textual action procedure" la definizione dei singoli passi della sequenza di interazione;
3. definire i vari step.
4. premere sul pulsante "Save" per aggiungere la nuova Macro Azione al progetto.

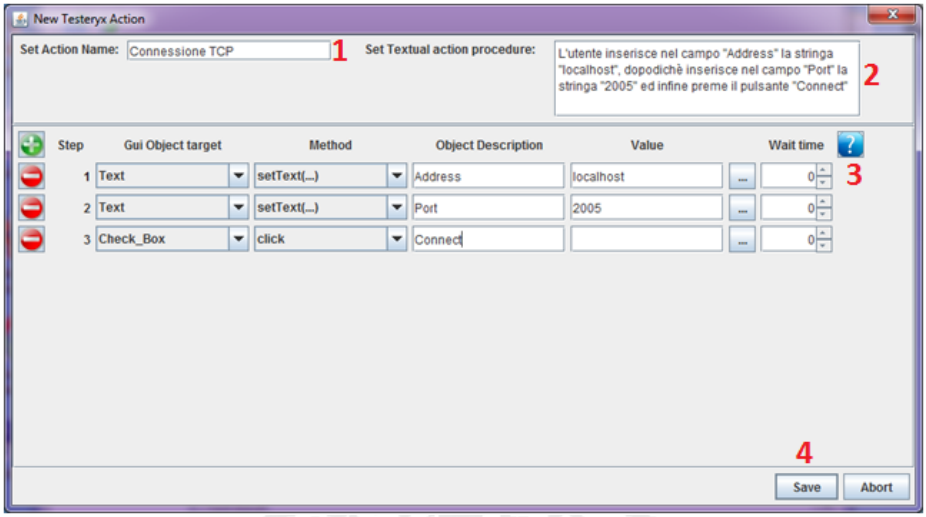

Figura 51 Definizione di Macro Azione.

Invece, per poter definire una Verifica occorre:

1. Premere il pulsante "new" nella colonna relativa alle Verifiche;
2. Definire il tipo di verifica da effettuare;
3. Premere sul pulsante "Save" per confermare.

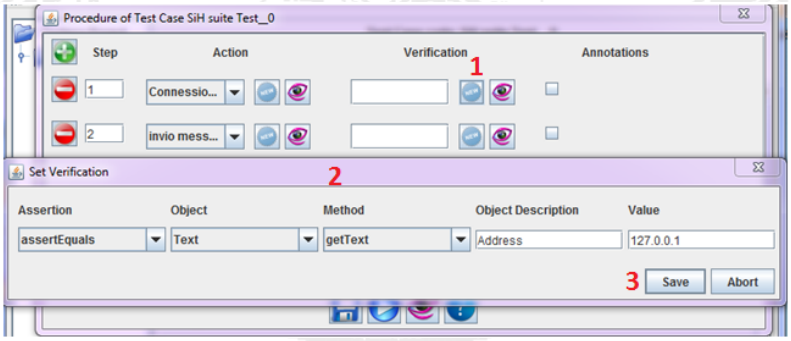

Figura 52 Definizione di un Assert.

B6. Esecuzione del Test Case

Per poter avviare l'esecuzione del Test Case occorre:

1. premere il pulsante "Run" dalla maschera di definizione della procedura;
2. indicare nel campo "Application Name" un nome mneumonico dell'applicazione da testare (opzionale);
3. nel campo "Jar file" selezionare il file .jar dell'applicazione *under test*
4. premere sul pulsante "OK" per avviare il Test.

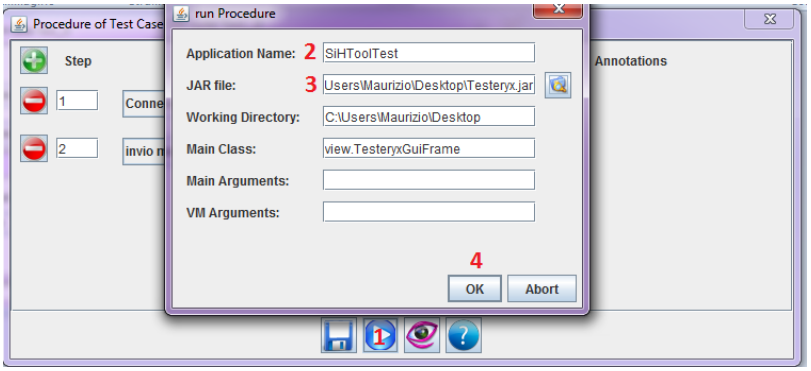

Figura 53 Pannello di esecuzione del TestCase.

B7. Testing di Regressione

Una volta definiti le procedure dei Test Case associati ad una Test Suite è possibile avviare l'esecuzione selettiva di essi automaticamente seguendo tale procedura:

1. Selezionare dal tree view il Test Suite desiderato.
2. Premere il pulsante "Run" dal pannel visualizzatosi successivamente al punto 1;
3. Selezionare i Test Case da dover eseguire
4. Premere il pulsante "Run" per eseguire le procedure di test desiderati.

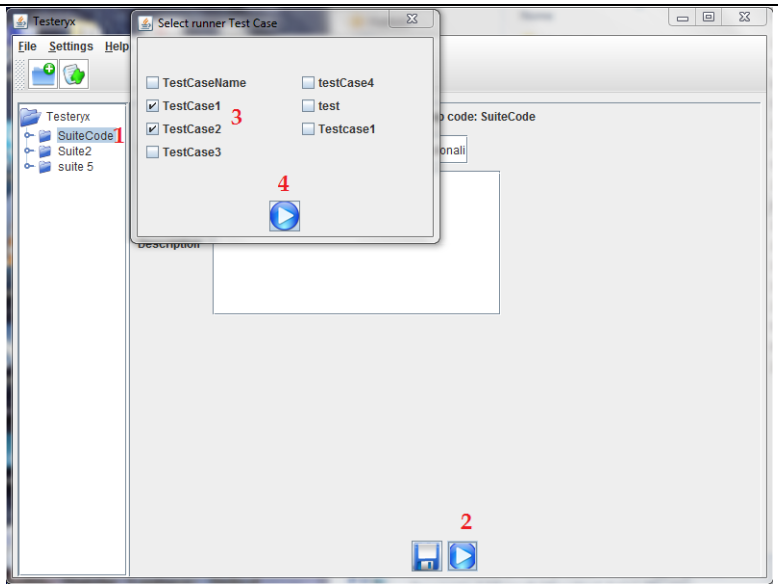

Figura 54 Esecuzione selettiva dei test case associati ad una Test Suite.

B8. Caricamento di un Testeryx Project

Per poter caricare un Testeryx Project occorre:

1. dal menù "File" premere su "Open Project"
2. selezionare la cartella in cui è contenuto il Testeryx Project da caricare;
3. premere su "Apri".

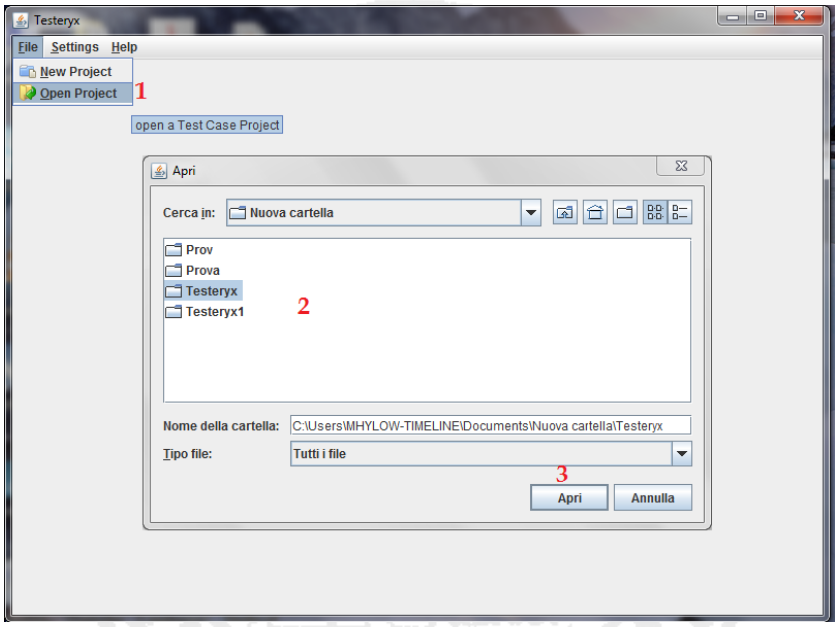

Figura 55 Procedura caricamento di un *Testeryx Project*.

Glossario

A

AWT = Abstract Window Toolkit

C

CSCI = Computer Software Configuration Item

CSU = Component Software Unit

D

DBMS = DataBase Management System

DTO = Data Transfer Object

F

FF. AA. = Forze Armate

FSM = Finite State Machine

G

GUI = Graphical User Interface

I

IEEE = Institute of Electrical and Electronics Engineers

M

MCP = Message Communication Protocol

MRM = Multiple Remote Messages

O

OO = Object Oriented

R

RCF = Remote Call Framework

S

SAN = System Abstract Name

SEM = Standard Excange Message

SRS = Software Requirements Specification

STD = Software Testing Description

STP = Software Test Plan

STR = Software Test Report

SUT = System Under Test

T

TCP = Trasmission Control Protocol

U

UAV = Unmanned Aerial Vehicle

UML = Unified Modeling Language

URL = Uniform Resoruce Locator

V

V&V = Verification and Validation

X

XML = eXtensible Markup Language

XSL = eXtensible Stylesheet Language

Bibliografia

[1] **Ian Sommervile**, *Software Engineering "Verifica e validazione"*. Pearson Education, 2007.

[2] **Bian Marick**, "The tester's triad: Bug, Product, User". Testing Foundation. 2000.

[3] **James Bach**, ″Test automation snake oil, v2.1." 1999.

[4] **Mauro Pezzè**, "Test ed analisi del software". 2000.

[5] **Brian Marick**, **Steve Stukenborg**, "The test manager at the project status meeting". 2011.

[6] **Brian Marick, Motorola Inc.** ″Experience with the cost of dirrente coverage goals for testing*".* 2001.

[7] **Brian Marick, James Back, Cem Kamer** "A manager's guide to evalueting test suites". 2002.

[8] **Brian Marick**, "When should a test be automated". 2002.

[9] **Cem Kaner**, "Architectures of test automation". 2003.

[10] **Cem Kaner**, "Quality cost analaysis: benefits ad risks". 2001.

[11] **James Bach**, "Exploratory testing and the planning myth". 2000.

[12] **Fewster.** "Software Test Automation: Effective Use of Test Execution Tools". Boston, Massachussetts, Addison-Wesley, 1999.

[13] **Williams, T. W., K. P. Parker.** "Design for Testability – A Survey." IEEE *Trans.* Comp. 31, 1982.

[14] **Pettichord B.** "Design for Testability" Pacific Northwest Software Quality Conference, 2001.

[15] **White L., H. Almezen.** "Generating test cases for GUI responsabilities using complete interaction sequences". San Jose, CA. 2000.

[16] **Meyer, S., L. Apfelbaum.** "Use Case Are Not Requirements". 1999.

[17] **Imbus, xx..** "Automated Testing Of Graphical User Interfaces (GUIs)". IEEE Software 2002.

[18] **Benedikt, M. J., Freire, P. Godefroid.** "Automatically Testing Dynamic Web Site". Bell Laboratories, Lucent Technologies, 2002.

[19] **Nyman, N.** "In Defence of Monkey Testing".Test Automation SIG Group, 2001.

[20] **Pekka Aho, Nadja Menz, Tomi Raty e Ina Schieferdecker.** "Automated Java GUI Modeling for Model-Based Testing Purposes". Eighth International Conference on Information Technology: New Generations, 2011.

[21] **Open source software testing tools,** http://www.opensourcetesting.org.

[22] **Cacique,** http://cacique.mercadolibre.com.

[23] **Concordion,** http://www.concordion.org.

[24] **Cucumber,** http://cukes.info.

[25] **GraphWalker,** http://graphwalker.org.

[26] **Harness,** http://sourceforge.net/project/harness/.

[27] **IdMUnit**, http://sourceforge.net/projects/idmunit/.

[28] **Ivalidator**, http://www.ivalidator.org.

[29] **Jamaleon,** http://jamaleon.sourceforge.net.

[30] **JDiffChaser,** http://jdiffchaser.sourceforge.net.

[31] **Jemmy,** http://java.net/project/jemmy.

[32] **Marathon,** http://marathontesting.com.

[33] **Maveryx,** http://www.maveryx.com**.**

[34] **Selex Sistemi Integrati**, http://www.selex-si.com.

Ringraziamenti

Ringrazio il Professor **Stefano Russo** innanzitutto per l'opportunità concessami nell'intraprendere questa splendida esperienza, grazie alla quale ho avuto modo di conoscere persone eccezionali e maturare metodologie di lavoro che da nessun libro avrei potuto trarre.

Ringrazio i miei due correlatori l'ing. **Gabriella Carrozza** e l'ing. **Roberto Pietrantuono** per l'avermi seguito e sostenuto durante le varie vicissitudini. Grazie a loro sostesto e professionalità tutto ciò che poteva sembrare inizialmente impossibile, a poco a poco diventava sempre più a portata di mano.

Ringrazio il **SESM** di Giugliano per l'avermi messo a disposizione le postazioni, le attrezzature, la loro professionalità e cortesia, grazie ai quali mi è stato possibile lavorare in un ambiente eccezionale.

Ringrazio il dott. **Antonio Divisano** per il supporto tecnico ma soprattutto interpersonale che si è creato non solo con lui ma con tutto il gruppo del SESM.

Ringrazio Alessio, Vittorio, Vincenzo, Sergio e ancora Antonio per le splendide giornate passate al SESM durante le allegre pause trascorse assieme.

Ringrazio Antonio Marotta e Nicola Traficante per essere stati dei validi amici/ compagni di studi durante il percorso della laurea specialistica.

Ringrazio mamma e papà per il sostegno economico ma soprattutto morale datomi durante l'intero percorso di studio. Ringrazio mio fratello Carmine e in mio

fratello Oreste per l'appoggio e la motivazione. Ringrazio mia zia Teresa e mio zio Valentino e i miei due cugini Andrea e Francesco per le allegre serate passate in allegria nei periodi di stress prima degli esami.

Ringrazio tutti i miei amici/colleghi di TeleContact e in particolar modo Marina Pa, Marina Pe, Domenico, Teresa, Alfredo, Ruggiero, Mario, etc....
Ringrazio gli amici conosciuti in questi anni di Università, che, in un modo o nell'altro, hanno contribuito al raggiungimento di questa meta, non faccio nomi per non dimenticarmi di nessuno.

Infine ringrazio la mia ragazza per avermi sopportato più che supportato durante tutti i miei sfoghi avuti nei momenti di difficoltà. Senza di lei non so se sarei stato capace di raggiungere tale obiettivo e so che grazie a lei riuscirò a raggiungerne degli altri.

Ancora Grazie a tutti.