

UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica



tesi di laurea specialistica

Sviluppo di un tool per il Testing di un sistema software distribuito

Anno Accademico 2007/2008

relatore

Ch.mo prof. Massimo Ficco

correlatore

Ing. Antonio Pecchia

candidato

Gabriele Gallo

matr. 885 / 57

*Alla mia famiglia.
In modo particolare al mio papà.*

Indice

Indice delle figure.....	VI
---------------------------------	-----------

Introduzione	VIII
---------------------------	-------------

Capitolo 1

Sistemi safety critical.....	11
-------------------------------------	-----------

1.1 Sistema, interfaccia e servizio.....	11
1.2 Il concetto di dependability.....	12
1.2.1 Attributi	12
1.2.2 Minacce	15
1.2.3 Mezzi.....	17
1.3 Critical system.....	18
1.3.1 Safety critical system	18
1.3.2 Dall'analisi al Testing dei safety critical system.....	19
1.3.2.1 Analisi dei fattori di rischio.....	20
1.3.2.2 Analisi requisiti di sicurezza	20
1.3.2.3 Progettazione	21
1.3.2.4 Testing	22
1.4 Sistemi per il controllo del traffico aereo (ATC).....	23
1.5 Il lavoro di tesi	25

Capitolo 2

Un middleware per l'ATC.....	27
-------------------------------------	-----------

2.1 Middleware per il traffico aereo	27
2.2 Requisiti di un sistema ATC	28
2.3 CORBA e ATC	30
2.4 Middleware utilizzato.....	33
2.5 Servizi offerti	34
2.5.1 System Managment	36
2.6 Ciclo di sviluppo: Modello a V	39

Capitolo 3

Testing della piattaforma middleware.....	42
3.1 Procedure di Testing	42
3.1.1 Test di unità.....	42
3.1.2 Test di validazione	44
3.1.3 Test di Qualifica.....	45
3.2 Analisi Test di Qualifica.....	46
3.2.1 La procedura di lancio	47
3.2.2 La procedura di valutazione.....	53
3.2.3 Problemi riscontrati.....	54

Capitolo 4

Dall'analisi alla progettazione del TOOL.....	55
4.1 Obiettivi del tool	55
4.2 Analisi e Specifica dei Requisiti	56
4.2.1 Individuazione dei casi d'uso	57
4.2.2 Specifica dei casi d'uso	57
4.2.2.1 Caso d'uso "AutomaticMode".....	58
4.2.2.2 Caso d'uso "ManualMode"	58
4.2.3 Sequence Diagram per i casi d'uso.....	59
4.3 Progettazione	61
4.3.1 Graphical User Interface	63
4.3.2 Core Application.....	65
4.3.3 Class Diagram.....	68

Capitolo 5

Implementazione del Tool "QUALTTOOL"	71
5.1 Individuazione di un pattern procedurale comune.....	71

5.2 Introduzione a “QUALTTOOL”	72
5.2.1 QUALTTOOL Interface	72
5.2.1.1 Manual Mode.....	74
5.2.1.2 Automatic Mode	79
5.2.2 Core Application.....	82
5.2.2.1 Avviamento di QUALTTOOL	83
5.2.2.2 Preparazione ambiente e avvio demoni	84
5.2.2.3 Servizi base per la preparazione dell’ambiente	85
5.2.2.4 Chiusura terminali e uccisione demoni	85
5.3 Sincronizzazione Demoni	86
5.4 Obiettivi conseguiti.....	87

Capitolo 6

Risultati Sperimentali.....88

6.1 Test Automatizzabili.....	88
6.2 Tempi di testing	93
6.3 Osservazioni sulla procedura di valutazione.....	93
6.3.1 Ispezione manuale.....	96
6.3.2 Proposte di soluzione.....	99

Conclusioni e sviluppi futuri.....100

Indice delle figure

Figura 1.1 Dependability.....	13
Figura 1.2 Binary state model	14
Figura 1.3 Sistema Fault-tolerant.....	15
Figura 1.4 Classificazione Guasti.....	16
Figura 1-5 Classificazione Failures.....	17
Figura 1.6 Catena fault-error-failure.....	17
Figura 1.7 Replicazione.....	18
Figura 1.8 Suddivisione delle responsabilità di un volo	25
Figura 2.1 Architettura di CORBA.....	31
Figura 2.2 CORBA Services e Facilities.....	32
Figura 2.3 Architettura generale del middleware.	34
Figura 2.4: Architettura del Core Middleware.....	35
Figura 2.5: servizi, tools e core.....	36
Figura 2.6 : processi del System Management.....	38
Figura 2.7 :Modello a V.....	40
Figura 3.1 Ambiente virtuale.....	43
Figura 3.2 Esecuzione test di qualifica.....	48
Figura 4.1 Diagramma dei casi d'uso.....	57
Figura 4.2 Sequence Diagram Automatic Mode	60
Figura 4.3 Sequence Diagram Manual Mode	61
Figura 4.4 Organizzazione progettuale ad alto livello.....	62
Figura 4.5 Interazione GUI-CORE	63
Figura 4.6 Interazioni User-GUI-Core	64
Figura 4.7 Class Diagram per la GUI	65

Figura 4.8 Use Case Diagram del Core	66
Figura 4.9 Class Diagram delCore.....	67
Figura 4.10 Sequence Diagram del Core	67
Figura 4.11 Class Diagram definitivo	68
Figura 5.1 Pattern	72
Figura 5.2 Interfaccia QUALTTOOL	75
Figura 5.3 Manual Mode... ..	76
Figura 5.4 File "config.xml"	77
Figura 5.5 Posizionamento automatico dei terminali di lavoro.....	79
Figura 5.6 AutomaticMode.....	80
Figura 5.7 Repository dei risultati.....	81
Figura 6.1 Esito di un Test	94
Figura 6.2 Valutazione dei log	96
Figura 6.3 Log platform daemon (term3).....	97
Figura 6.4 Log FT_manager daemon.....	97
Figura 6.5 Log Observer daemon.....	98
Figura 6.6 Log Supervision daemon	98

Introduzione

Nel contesto dei sistemi per il controllo del traffico aereo (ATC) il concetto di **rischio** (*hazard*) assume una notevole importanza, in quanto l'occorrenza di fallimenti può comportare gravi danni a persone o all'ambiente circostante. Nonostante l'evidente gravità delle possibili conseguenze, la complessità di questi sistemi cresce a ritmi sostenuti, incrementando evidentemente le difficoltà di gestione degli stessi e le probabilità che essi possano fallire in maniera catastrofica.

Sistemi in cui un fallimento può provocare la perdita di vite umane, o danni di paragonabile ingenza, sono noti in letteratura come "*safety critical systems*". Occorre quindi progettare sistemi sicuri che riducano il rischio di incidenti e tali sistemi sono caratterizzati da un ciclo di sviluppo lungo e rigoroso e dalla necessità di utilizzare soluzioni software distribuite per soddisfarne i requisiti.

Una delle fasi fondamentali nello sviluppo di sistemi sicuri è la fase di testing, ossia quell'attività di "esercizio" del software tesa all'individuazione dei malfunzionamenti prima della messa in esercizio.

Il presente lavoro di tesi è stato condotto nell'ambito del progetto COSMIC, laboratorio pubblico-privato dei partner CINI, DIS-UNINA, CRIAI, SELEX-SI e SESM.

Focalizza l'attenzione sulla fase di testing di tali sistemi, in particolare sui test di qualifica di una piattaforma middleware (*Open Source*) che presenta le seguenti caratteristiche :

- è un sistema safety critical;
- è un sistema mission critical;
- è un middleware Corba-based;
- organizza i proprio servizi in moduli (CSCI).

➤ Cerca di soddisfare i requisiti dei sistemi per l'ATC

Analizzeremo tale attività, riscontrandone complessità ed onerosità. Quindi proporremo uno strumento in grado di facilitare la fase di testing, e cercheremo di ridurre interazione con utente e i tempi necessari per l'attività di testing.

Obiettivo principale è automatizzare tutta l'attività necessaria alle fasi di preparazione e avvio dei test.

La tesi è articolata in 6 capitoli il cui contenuto può essere così riassunto :

1. Il primo capitolo fornisce una sommaria descrizione dei “safety critical systems”, dei concetti ad essa correlati ed introduce gli obiettivi che si intende raggiungere;
2. Il secondo capitolo presenta una definizione della piattaforma middleware su cui si basa il progetto COSMIC. Ne presentiamo le principali caratteristiche, e concludiamo con una descrizione del suo modello di sviluppo, con particolare attenzione alla fase di testing.
3. Il terzo capitolo mostra com'è schematizzata l'attività di testing della piattaforma per poi soffermarsi sulle operazioni necessarie all'esecuzione di un singolo test di qualifica; descrive infine i limiti di tale procedura di testing, limiti che il tool da sviluppare dovrà superare;
4. Il quarto capitolo evidenzia le motivazioni che hanno portato allo sviluppo del tool per il testing di un sistema software distribuito dando una descrizione completa di tutta la fase di progettazione necessaria a raggiungere gli obiettivi prefissati;

5. Il quinto capitolo riguarda gli aspetti implementativi del tool, ossia la sua organizzazione interna e le varie scelte effettuate per soddisfare i requisiti evidenziati in fase di analisi.
6. Il sesto capitolo presenta in maniera dettagliata i risultati sperimentali conseguiti e riporta ulteriori osservazioni relative alla valutazione dei test.



Capitolo 1

Sistemi Safety critical

In questo capitolo vengono introdotti alcuni concetti di base e vengono presentati i sistemi safety critical. Si discuterà di uno dei principali contesti applicativi come i sistemi per il controllo del traffico aereo (ATC).

Introduzione

Negli ultimi anni i sistemi informatici sono stati utilizzati in scenari critici dal punto di vista economico, sia in termini di affidabilità che di sicurezza. A tal proposito, possiamo pensare a sistemi per il controllo del traffico aereo, per il controllo ferroviario, a sistemi di e-commerce e a molte altre applicazioni. In questi contesti è diventato fondamentale fornire il funzionamento del sistema in presenza di malfunzionamenti e garantire l'assenza di danni a persone e/o all'ambiente operativo.

La realizzazione di un sistema con tali caratteristiche richiede costi e tempi di sviluppo notevoli. Tutte le fasi del ciclo di sviluppo vengono influenzate ed in particolare poniamo la nostra attenzione su quella di testing.

La fase di testing di un sistema del genere è molto complessa ed onerosa, e questo lavoro di tesi si pone proprio l'obiettivo di automatizzare tale fase rendendola meno costosa sia dal punto di vista economico che da quello temporale.

1.1 Sistema, interfaccia, servizio

Un sistema è una qualsiasi entità in grado di interagire con altre entità, persone o altri sistemi. Realizzato per fornire un certo numero di servizi attraverso la sua interfaccia, che ne delimita i confini con il mondo esterno. Un servizio non è altro che un comportamento del sistema così com'è percepito dall'utente. Si dice **corretto** se conforme alle specifiche, altrimenti si parla di **failure**

Un sistema si dice **safety-critical** se un malfunzionamento può causare danni a persone e/o all'ambiente circostante. Si definisce **mission-critical**, un sistema in cui un malfunzionamento può compromettere il raggiungimento di un obiettivo voluto.

1.2 Il concetto di dependability

Non è semplice definire la dependability, in quanto vi sono una serie di nozioni che vanno ben oltre al concetto di affidabilità. Vi sono varie definizioni di dependability, quella forse più completa è la seguente: *la capacità di un sistema di fornire un servizio su cui possa essere riposta giustificata fiducia.*

Per poter descrivere in modo corretto il concetto di dependability è necessario seguire un'approccio sistematico, parlando di attributi, mezzi e minacce.

1.2.1 Attributi

La dependability è un macroattributo, in cui confluiscono i seguenti attributi:



Figura1-1 Dependability

- **Availability:** un sistema è available se, in un dato istante t è in grado di fornire un servizio corretto.

$$A(t) = P(\text{!Failure in } t)$$

L'Availability è interpretata come un valore medio, ossia come la probabilità che in un dato istante il sistema sia UP o DOWN

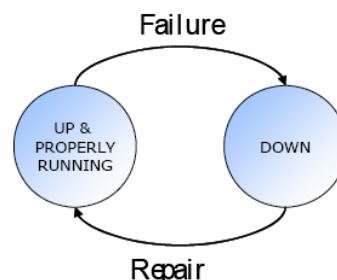


Figura 1-2 Binary state model

- **Reliability:** misura dell'intervallo di tempo in cui il sistema fornisce un servizio corretto in modo continuativo.

$$R(t) = P(\text{!Failure in } (0,t))$$

- **Safety:** assenza di condizioni di funzionamento che possono arrecare danni a persone e/o cose. Definiti i fallimenti catastrofici, attraverso una valutazione dei rischi:

$$S(t) = P(\text{!CatastrophicFailure in } (0,t))$$

Matematicamente la funzione *safety* $S(t)$ è la probabilità che non vi siano guasti catastrofici in $[0, t]$.

- **Performability:** metrica per definire le prestazioni del sistema, anche in caso di guasto. Nel contesto di sistemi tolleranti ai guasti (sistemi *fault-tolerant*), risulta inadeguato un modello degli stati binario come quello di figura . Sistemi del genere infatti, seppure in condizioni di performance degradate, riescono ad operare anche in presenza di fallimenti. Il diagramma degli stati per un sistema fault-tolerant avrà quindi un numero di stati pari almeno al numero di failure che è in grado di tollerare.

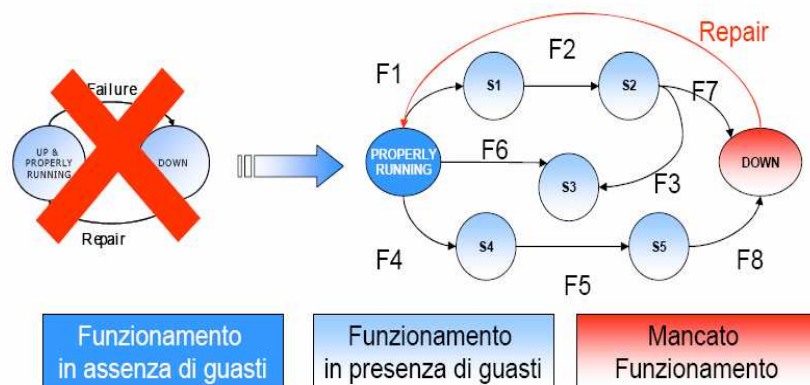


Figura 1-3 Sistema Fault-tolerant

- **Maintainability:** capacità di un sistema di essere sottoposto facilmente a modifiche e/o riparazioni;
- **Security:** è un attributo *composto* associato all'assenza di manipolazioni improprie ed accessi non autorizzati al sistema. E' suddiviso in tre sottoproprietà:
 - **Availability:** disponibilità del sistema esclusivamente per gli utenti autorizzati

- **Integrità:** prevenzione da alterazioni improprie dello stato del sistema
- **Confidentiality:** prevenzione dalla diffusione non autorizzata di informazioni.

1.2.2 Minacce

Un sistema può fallire per molteplici cause. Le più comuni sono guasti hardware, errori di progettazione hardware o software e, ancora, errati interventi di manutenzione.

Per parlarne è necessario introdurre i concetti di fault, error, failure.

- **fault:** stato improprio dell'hardware o del software del sistema, causato dal guasto di un componente, da fenomeni di interferenza o da errori di progettazione. I fault sono molteplici e, tipicamente, sono classificati in accordo alla figura ;

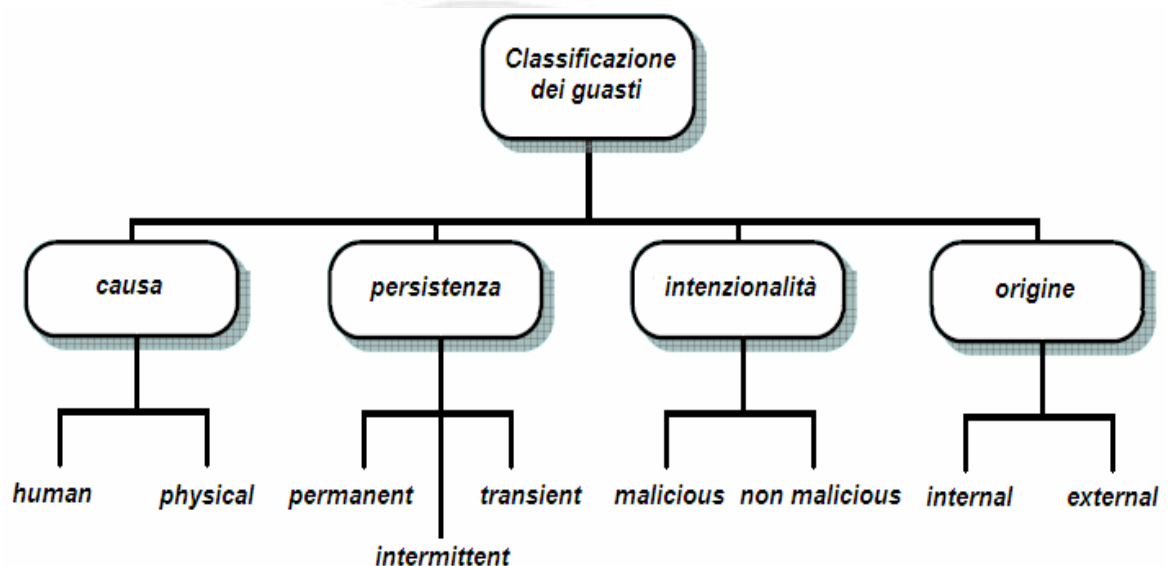


Figura 1-4 Classificazione Guasti

- **error:** parte dello stato di un sistema che può indurre al fallimento. La causa di un errore è un fault. Uno stesso fault, può generare più errori (*multiple*

related error);

- **failure**: evento in corrispondenza del quale un sistema cessa di fornire un servizio corretto. I failure sono molteplici e possono essere classificati in accordo alla tassonomia di figura ;

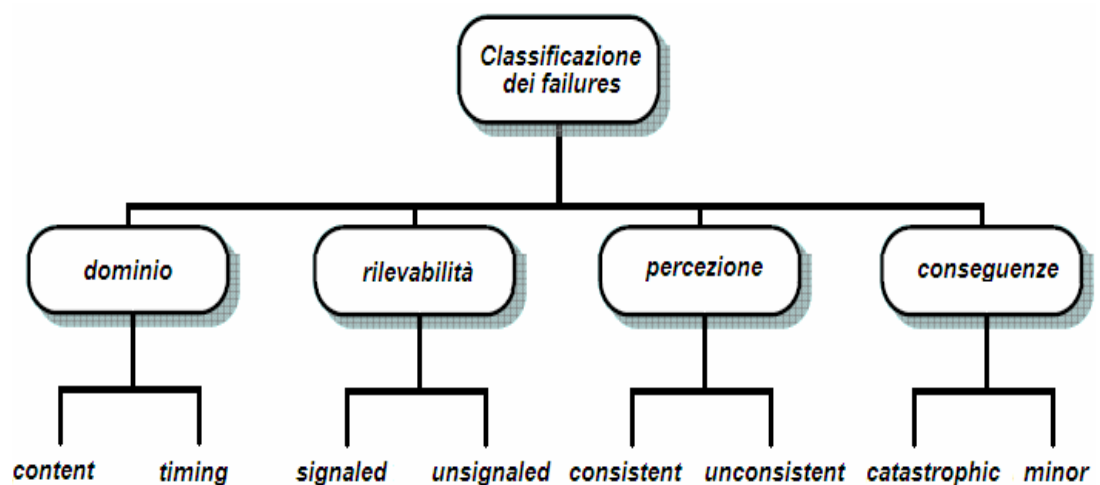


Figura 1-5 Classificazione Failures

Le minacce che possono condurre al fallimento di un sistema, si propagano secondo uno schema ben preciso(**catena fault-error-failure**): l'attivazione di un guasto (*fault*), causa la transizione da uno stato di corretto funzionamento ad uno improprio (*error*). Un error può generare un *failure* quando raggiunge l'interfaccia del sistema.

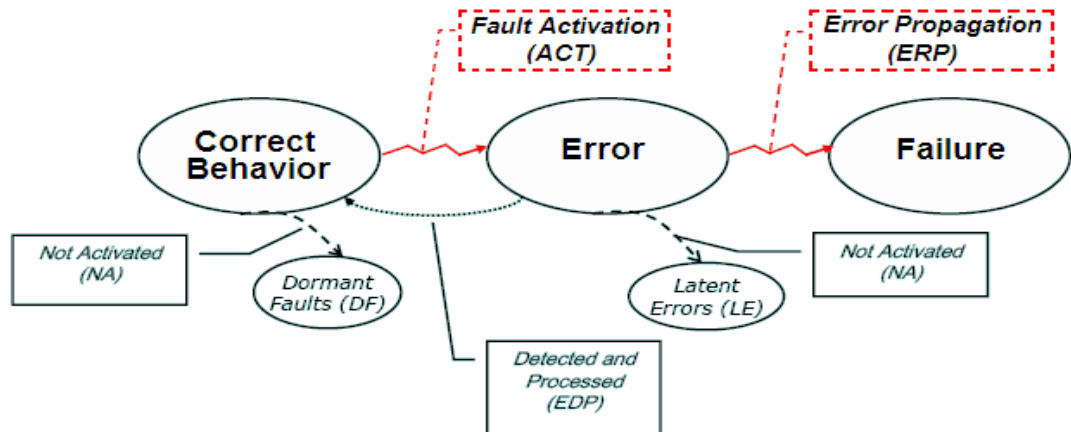


Figura 1-6 Catena fault-error-failure

1.2.3 Mezzi

Con il termine mezzi, vengono indicate le tecniche capaci di incrementare il grado di dependability di un sistema. La scelta del particolare approccio dipende dalla tipologia di sistema e dallo specifico attributo che si vuole migliorare.

Di seguito, vengono brevemente presentati i principali *means*(*mezzi*) utilizzati in pratica:

- **fault avoidance:** tecniche orientate a minimizzare la probabilità di occorrenza dei fallimenti. Tali tecniche, implicano l'utilizzo di componenti altamente affidabili che, pertanto, comportano un incremento dei costi;
- **fault tolerance:** tecniche orientate alla minimizzazione delle conseguenze dei guasti e che tendono ad evitare che possano degenerare in un failure .
Tipicamente,esse si articolano in due fasi (error *detection* ed error *treatment* con annesso *system recovery*) ed impiegano approcci basati sulla replicazione (*spaziale* o *temporale*);

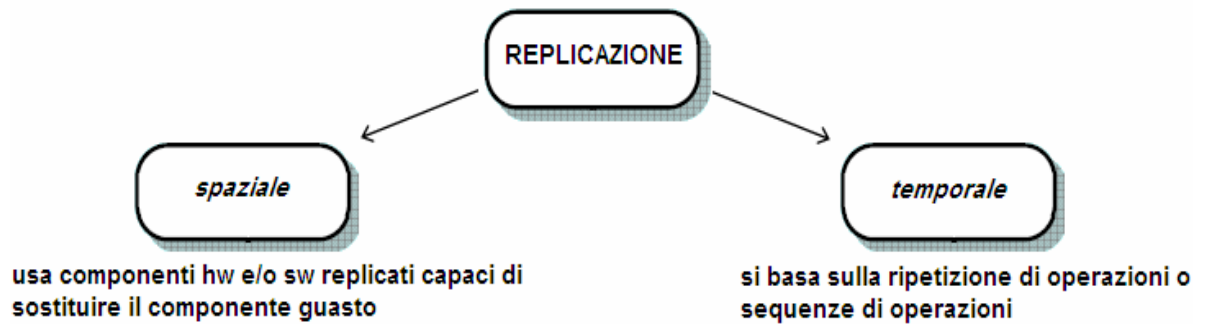


Figura 1-7 Replicazione

- **fault removal:** tecniche orientate alla individuazione degli errori e alla rimozione dei guasti durante lo sviluppo o in fase operativa;
- **fault forecasting:** consiste nella stima del numero corrente, dell'incidenza e delle conseguenze dei fault. Può essere
 - *deterministico* : studio degli effetti dei guasti sul sistema.
 - *probabilistico*: stima dei parametri di dependability .

1.3 Critical System

I "Critical System" possono essere classificati, in base allo scopo e al dominio applicativo:

- **Safety-critical:** sistemi il cui fallimento (failure) può causare ferimenti, perdite di vita, o seri danni ambientali (e.g. sistema di controllo per un impianto chimico);
- **Mission-critical:** sistemi il cui fallimento (failure) può causare il fallimento di attività guidate da obiettivo (e.g. sistema di navigazione di una navetta spaziale);
- **Business-critical:** sistemi il cui fallimento (failure) può causare ingenti perdite di denaro (e.g. sistema di gestione dei conti correnti in una banca).

1.3.1 Safety Critical System

In contesti applicativi in cui è messa in pericolo la vita umana, il concetto di **rischio** (*hazard*) assume una notevole importanza. Nonostante ciò, la complessità dei sistemi cresce a ritmi sostenuti, incrementando evidentemente le difficoltà di gestione degli stessi e le probabilità che essi possano fallire in maniera catastrofica. Il *Dipartimento di Difesa* degli Stati Uniti d'America definisce come *mishap* un evento, o una serie di eventi non previsti, che possono causare danni fisici a persone o compromettere l'ambiente circostante. Un sistema *safety critical* è caratterizzato, pertanto, dalla probabilità di occorrenza di un mishap.

E' evidente che una riduzione dei rischi, ovvero la minimizzazione delle probabilità di occorrenza di fallimenti catastrofici, comporta un aumento dei costi, tirando in gioco anche aspetti economici nella gestione di un sistema critico. Occorre stabilire in quale misura la riduzione dei rischi può ritenersi accettabile.

Considerato che la tecnologia è sempre più controllata dal software, una buona parte dei rischi è rimessa nelle mani degli ingegneri del software, in quanto la qualità del software da essi prodotto ne influenza sensibilmente l'occorrenza. Nella maggior parte dei casi un rischio non produce alcun tipo di effetto; tuttavia nel peggiore dei casi questo è in grado di produrre conseguenze catastrofiche capaci di minare la salute, o in certi casi, la vita di esseri umani. Pertanto appare evidente come, in tali ambienti, il software assuma un ruolo fondamentale nonché critico.

Si consideri che ad oggi il software costituisce un componente imprescindibile in sistemi di controllo nucleari, negli autoveicoli e nei sistemi per la difesa militare e per il controllo del traffico aereo. In questi contesti un rischio può essere definito come un insieme di condizioni ambientali e di eventi che possono causare un incidente.

In un sistema software, inoltre, un difetto può risultare dormiente, e dunque non rilevabile, finché un particolare insieme di condizioni operative non ne provoca la

manifestazione.

1.3.2 Dall'analisi al testing dei safety critical system

La dipendenza dei “*safety critical system*” dal software può risultare controproducente , in quanto il software contribuisce ad aumentare la sicurezza del sistema, ma può anche, portare il sistema in uno stato instabile e pericoloso.

Pertanto la progettazione di sistemi sicuri segue il “*software engineering for safety*” che cerca di ridurre il rischio di incidenti. Tale approccio è articolato in :

- analisi dei fattori di rischi;
- analisi e specifica dei requisiti di sicurezza;
- progettazione rivolta alla sicurezza;
- testing;

1.3.2.1 Analisi dei fattori di rischio

L'individuazione dei rischi ed una loro successiva analisi risultano indispensabili per la sicurezza del sistema; tali operazioni di analisi vengono effettuate in base al livello di criticità ed alla probabilità di occorrenza del rischio.

Il primo passo è l'individuazione dei rischi che possono essere eliminati da particolari scelte progettuali, detti infatti “rischi eliminabili”. Successivamente si individuano, tramite tecniche particolari quali il “criticality analysis” ed il “fault tree analysis”, i componenti software che contribuiscono all'esistenza od alla prevenzione di ogni rischio.

La gestione dei rischi può variare da sistema a sistema; può, infatti, essere richiesto di prevenire il verificarsi di situazioni pericolose (es. tramite la mutua esclusione o i timeout), oppure semplicemente di individuare una situazione pericolosa, o di portare il sistema da uno stato pericoloso ad uno sicuro.

1.3.2.2 Analisi e specifica dei requisiti di sicurezza

L'individuazione formale dei requisiti migliora la qualità del prodotto finale in quanto facilita e rende più accurata la progettazione, l'implementazione e lo sviluppo dei casi di test; l'analisi formale, inoltre, consente di determinare mediante meccanismi automatici se sono preservate le proprietà di sicurezza e se una combinazione di eventi può portare il sistema in uno stato instabile.

Per garantire la sicurezza del sistema, inoltre, devono essere annullate eventuali discordanze che possono insistere tra i requisiti di sicurezza del sistema ed i requisiti del software. In quest'ottica, quindi, una corretta individuazione dei requisiti di sicurezza diventa fondamentale al fine di sviluppare sistemi software sicuri; dato che un minimo errore in fase di analisi dei requisiti può causare degli inconvenienti, essi non solo devono essere identificati e specificati, ma devono anche essere sottoposti a verifiche accurate.

1.3.2.3 - Progettazione rivolta alla sicurezza

Un sistema software progettato per la sicurezza può gestire i pericoli in diversi modi: può impedirli, oppure individuarli e controllarli. La progettazione rivolta alla prevenzione dei rischi (impedire i pericoli) prevede meccanismi come "hardware lockouts", "lockins", "interlocks", "watchdog timers", l'isolamento dei moduli "safety critical" e che il comportamento del sistema sia quello atteso.

L'individuazione ed il controllo dei pericoli, invece, prevede meccanismi come la

progettazione a sicurezza intrinseca, test automatici, gestione delle eccezioni, controlli su errori degli utenti e riconfigurazione.

Inoltre, la progettazione di sistemi sicuri introduce tre problematiche principali:

1. **Design tradeoff**: la progettazione dei safety system deve garantire non solo lo sviluppo dei requisiti di sicurezza, ma anche che le proprietà del sistema siano soddisfatte.
2. **Vulnerabilità ai semplici errori progettuali**: si tende a pensare che i problemi di progettazione rivolta alla sicurezza siano tutti di alta complessità; ciò in realtà non è vero, dato che sono numerosi gli incidenti causati da errori semplici, talvolta noti, spesso facilmente prevenibili in fase di progettazione o semplici da individuare in fase di testing. Non è vero, tuttavia, che da piccoli errori derivino piccole conseguenze; gli effetti degli errori in un software non sono mai prevedibili a priori.
3. **Uso di tecniche progettuali note**: nella progettazione dei safety system è sempre opportuno seguire le tecniche progettuali note; in tal modo si evita di incappare in errori che siano già stati commessi in precedenza e poi corretti mediante l'uso di tecniche formali di progettazione.

In definitiva, per ridurre il rischio di mishap possiamo utilizzare vari approcci tra i quali :

- migliorare l'affidabilità e la qualità dei componenti;
- utilizzare dispositivi interni per la sicurezza;
- utilizzare dispositivi esterni per la sicurezza.

1.3.2.4 – Testing

Il testing verifica che i requisiti di sicurezza del software siano soddisfatti dal sistema e

che l'analisi dei rischi sia corretta.

I requisiti di sicurezza spesso descrivono condizioni invarianti che devono verificarsi in tutte le circostanze; il testing in questi casi verifica che il sistema sia tollerante al verificarsi di eventuali errori del software ("fault-tolerance"); il testing, inoltre, può anche dimostrare che il software risponda appropriatamente a situazioni anomale. A tal proposito i casi di test devono marcare le condizioni di inizio e di fine (startup e shutdown) e le condizioni anomale (individuazione e correzione degli errori), poiché da una gestione impropria di questi stati vulnerabili possono derivare dei mishap.

Il testing di un sistema sicuro viene notevolmente complicato da problematiche legate al:

- **Dominio.** Un sistema può risultare pericoloso a causa di errati presupposti sul dominio nel quale il sistema stesso opera; ciò può accadere per esempio nello sviluppo di software per mezzi spaziali in cui è facile effettuare valutazioni errate sull'ambiente. Le incertezze riguardanti il dominio di applicazione complicano l'individuazione del punto esatto in cui il sistema muta; esso può, infatti, mutare in uno stato non sicuro, complicando le eventuali correzioni in grado di riportarlo in uno stato sicuro. Una giusta modellazione del dominio di tali sistemi risulta quindi essenziale per una corretta determinazione dei casi di test.
- **Utente.** Allo stesso modo assunzioni errate sull'utente o sull'operatore contribuiscono a rendere un sistema non sicuro; un utente può, ad esempio, utilizzare il sistema in modo non corretto, compromettendone quindi la sicurezza. Pertanto i fattori umani sono da non sottovalutare, ed è solo nel corso del testing che tali discordanze vengono alla luce.

1.4 - Sistemi per il controllo del traffico aereo (ATC)

Un esempio significativo di “safety critical system” è il sistema di Controllo del Traffico Aereo (ATC).

In base alle direttive dell' Organizzazione Aviazione Civile Internazionale (ICAO), ai servizi del traffico aereo sono demandati i seguenti compiti:

- Prevenire le collisioni tra gli aeromobili durante la fase di volo;
- Prevenire le collisioni tra aeromobili ed eventuali ostacoli durante la fase di manovra;
- Regolare e accelerare il traffico aereo;
- Fornire informazioni e avvisi per garantire l'efficienza e la sicurezza dei voli;
- Allertare, in base alla necessità delle richieste pervenute, gli organismi previsti per

i servizi di ispezione e soccorso, e fornire ad essi un adeguato supporto.

Un aeromobile che voglia attraversare lo spazio aereo, si tratti di compagnie aeree o di privati, deve sottoporre all'attenzione dell' Ente Nazionale di Assistenza al Volo (ENAV) il relativo piano di volo; se approvato l'aeromobile è pronto a partire.

Con riferimento ad un generico volo civile con partenza da Roma Fiumicino ed arrivo a Milano Malpensa, come illustrato in figura , le responsabilità del volo sono suddivise nel seguente modo.

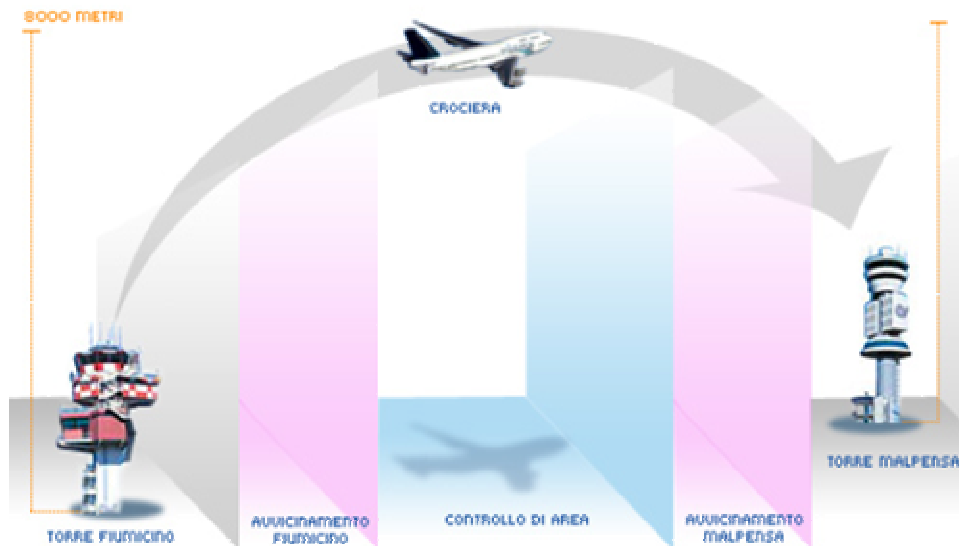


Figura 1-8 Suddivisione delle responsabilità di un volo

Alla Torre di Controllo è affidata la garanzia della sicurezza di un aeromobile nell'aeroporto e nelle sue vicinanze. La gestione dei movimenti a terra, che includono l'allontanamento dai parcheggi e il rullaggio, è di competenza del controllore di terra, che, esaurito il suo compito, affida il volo al controllore di torre, cui spetta concedere l'autorizzazione al decollo in modo che sia garantita la distanza di sicurezza da tutti gli altri aeromobili.

All' avvicinamento spetta poi il compito di instradare i velivoli appena decollati facendoli salire per l'inserimento in aerovia. La gestione passa quindi al centro di controllo d'area, che ha il compito di armonizzare l'ingresso e l'uscita degli aeromobili con il grande flusso

di traffico che scorre lungo le aerovie.

Ai velivoli vengono assegnati differenti livelli di volo e indicate le traiettorie da seguire,

al fine di rimanere sempre reciprocamente distanziati fino a che verranno affidati nuovamente al controllore di destinazione che ne gestirà l'avvicinamento. In tale fase si stabilisce la corretta sequenza degli aeromobili e provvede a fornire una guida quando lasciano le aerovie per iniziare la discesa fino all'allineamento con la pista. Quando il velivolo è ormai stabilizzato sul sentiero di atterraggio ed in vista dell'aeroporto, la gestione viene affidata alla torre di controllo di destinazione, che scorta l'aereo fino al parcheggio. Nel caso degli aeromobili che entrano o escono dallo spazio aereo italiano i controllori e i sistemi informatici dialogano costantemente con gli enti stranieri limitrofi inviando e ricevendo notizie sui voli che permettano all'ente interessato di prepararsi alla gestione.

1.5 Il lavoro di tesi

La piattaforma middleware oggetto del seguente lavoro di tesi è un chiaro esempio di “safety and mission critical system”.

L'attenzione principale è rivolta all'attività di esecuzione dei test di qualifica di tale piattaforma; vedremo che tale operazione è onerosa e complessa e richiede una continua interazione utente-macchina. Obiettivo principale è quello di fornire all'utente uno strumento semplice e rapido in grado di eseguire in maniera automatica tutta la fase di preparazione e avvio dei test. Vedremo infatti che lo sviluppo di tale strumento offre all'utente notevoli vantaggi sia in termini di tempo che di facilità di esecuzione dell'attività di testing. Presenteremo quindi innanzitutto una breve descrizione del testing della piattaforma middleware in esame per poi analizzarne ed evidenziarne i limiti; progetteremo quindi un tool con lo scopo di superare tali limiti.

Capitolo 2

Un middleware per l'ATC

2.1 Middleware per il traffico aereo

Come accenato nel capitolo 1, i sistemi ATC sono complessi, real-time, *safety-critical* e pertanto richiedono in genere tempi di sviluppo lunghi. Molti sistemi ATC utilizzano infrastrutture specializzate proprietarie sia hardware sia software che pongono non pochi problemi per le operazioni di aggiornamento e per il soddisfacimento delle sempre più crescenti esigenze del dominio applicativo. Negli ultimi tempi si sta diffondendo l'impiego di tecnologie orientate agli oggetti e al calcolo distribuito, il riutilizzo del codice, l'utilizzo massiccio dei *Commercial OFF-The-Shelf* (COTS), che permettono di ridurre notevolmente i tempi e costi di sviluppo di sistemi ATC. In particolare il modello a oggetti distribuiti permette di migliorare:

- la scalabilità (attraverso la modularità);
- l'estensibilità (attraverso fasi dinamiche di configurazione e riconfigurazione);
- le performance (attraverso l'elaborazione parallela);
- l'*availability* (attraverso la ridondanza);
- la riduzione dei costi (attraverso la condivisione di risorse).

Tuttavia il modello a oggetti distribuiti non presenta ancora alcune caratteristiche che sono invece essenziali in generale per i sistemi *safety* e *mission critical*, in particolare per l'ATC.

Di seguito saranno evidenziati vantaggi e svantaggi legati all'impiego di tecnologie basate sul modello a oggetti distribuiti per lo sviluppo di sistemi ATC.

2.2 Requisiti di un sistema ATC

Una tipica applicazione ATC prevede determinati requisiti, che in linea generale sono condivisi con altri scenari critici.

- **Availability.** Un sistema ATC richiede tipicamente valori alti di *availability*. Ad esempio alcuni componenti del sistema hanno come requisito un valore di *availability* superiore a 0.999995, a cui corrisponde 2.6 minuti di assenza di disponibilità per anno. L'*availability* di un sistema influenza pesantemente la progettazione dell'architettura software e hardware. Tale requisito viene comunemente soddisfatto attraverso la ridondanza. In genere è previsto che le parti *serventi* dell'applicazione vengano replicate, in maniera tale che il sistema continui a funzionare anche in presenza di guasti. Ovviamente una problematica legata alla replicazione è la gestione della consistenza dello stato, ovvero garantire l'allineamento dei dati sulle diverse repliche.
- **Paradigma di comunicazione.** Un' applicazione ATC utilizza in genere due paradigmi di comunicazione:
 1. *Client/Server*. Il client è l'entità che genera una richiesta verso il server .Il server risponde inviando la risposta della richiesta. Al fine di ottenere livelli maggiori di concorrenza, è preferibile prevedere la comunicazione di tipo

asincrono così che il client mentre è in attesa di risposta può continuare la relativa elaborazione.

2. *Publish/Subscribe*. I processi produttori pubblicano messaggi differenziati per tipo (*publish*), e i consumatori possono dichiararsi interessati ai messaggi in base al loro tipo (*subscribe*), al fine di ricevere dal middleware una notifica della presenza nella coda dei messaggi di interesse. Un determinato tipo di dato può avere uno o più *publisher*. L'interfaccia *publish/subscribe* permette al *subscribe* di richiedere un evento iniziale (*reconstitution*), per ricevere lo stato corrente dei dati. Un *subscribe* può cancellare la propria registrazione in ogni istante.

➤ **Qualità del servizio di comunicazione.** Un applicazione ATC ha come requisito i seguenti accoppiamenti meccanismo di comunicazione / livello di qualità:

- ✓ ***Reliable Point-to-Point***. Un singolo messaggio viene inviato ad un singolo destinatario. La consegna del messaggio deve essere garantita mentre un eventuale fallimento deve essere rilevato in modalità sincrona. Tale servizio è necessario per la comunicazione tra i client dotati di interfaccia HMI che richiedono servizi o dati ai processi server.
- ✓ ***Reliable Multicast***. Un singolo messaggio viene inviato a destinatari multipli. La consegna del messaggio deve essere garantita mentre un eventuale fallimento deve essere rilevato in modalità sincrona. Tale servizio è necessario per l'invio di dati critici, come ad esempio i piani di volo.

- ✓ **Unreliable Multicast.** Un singolo messaggio viene inviato a destinatari multipli.

La consegna del messaggio non deve essere garantita. Tale servizio è necessario per l'invio di messaggi caratterizzati da un' elevata mole di dati.

- **Conversione di dati.** Nello scenario ATC è possibile che sia necessario lo scambio di dati tra sistemi eterogenei (ad esempio tra sistemi che prevedono la codifica *little* e *big endian*). Occorre quindi che il middleware fornisca dei metodi per convertire i dati in un formato standard.
- **Selezione e localizzazione degli oggetti.** Le applicazioni ATC richiedono il servizio dei nomi che permette all'entità client di accedere all'oggetto remoto attraverso un nome piuttosto che attraverso la coppia IP/porto. Il servizio dei nomi diventa necessario nel contesto di servizi replicati, che quindi possono subire cambiamenti e riconfigurazioni nel tempo nel caso in cui si verificano dei guasti.
- **Real-Time.** I sistemi ATC vengono tipicamente classificati come sistemi *soft real-time* con specifici tempi di risposta *end-to-end*. Questa proprietà consente che la *deadline* non venga rispettata in qualche occasione e che il servizio offerto risulti occasionalmente deteriorato.

2.3 CORBA e ATC

CORBA, acronimo di *Common Object Request Broker Architecture*, è la specifica di un modello di riferimento per un middleware a oggetti, che fa parte di un più ampio modello architetturale denominato *Object Management Architecture* (OMA). CORBA è definito dal consorzio Object Management Group (OMG) formatosi nel 1989. CORBA costituisce un

middleware che ben si adatta nel contesto dell'ATC. Infatti esso, in quanto standard industriale per i sistemi distribuiti orientati agli oggetti, permette:

- Di separare la definizione di un oggetto dalla relativa implementazione, permettendo di sviluppare il *legacy code* (ovverò favorisce l'ereditarietà del codice) e di utilizzare componenti ATC di altri venditori.
- Di favorire la portabilità del codice grazie alla definizione di un'interfaccia indipendente dalla particolare piattaforma.

La seguente figura fornisce una vista semplificata dell'architettura di CORBA.

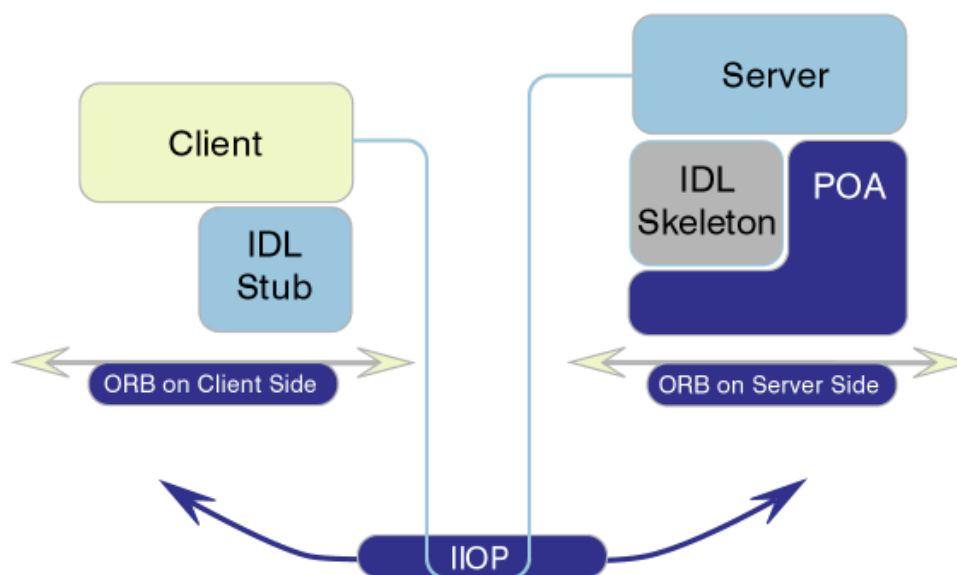


Figura 2.1 Architettura di CORBA

L' *Object Request Broker* (ORB) è l'elemento chiave che permette agli oggetti client e server di interagire indipendentemente dalla localizzazione degli oggetti, dal linguaggio e dalla piattaforma sottostante. L' ORB del client e del server sono collegati attraverso una serie di interfacce scritte in IDL (*Interface Definition Language*). In aggiunta CORBA definisce i *Common Services* e le *Common Facilities*. I Common Services sono un insieme di servizi a livello di sistema con interfacce specificate in IDL che estendono o completano

le funzionalità dell'ORB. Lo standard definisce 15 tipologie di servizi, quelli maggiormente usati nel contesto dell'ATC sono:

Event Service. Esso fornisce all'ATC le interfacce publish/subscribe.

Naming Service. Esso fornisce agli oggetti meccanismi necessari alla loro localizzazione e di mantenere l'associazione con un nome.

Time Service. Esso fornisce interfacce per la sincronizzazione in un ambiente di oggetti distribuiti.

Transaction Service. Esso fornisce agli oggetti distribuiti la possibilità di utilizzo del protocollo di *commit* a due fasi, che può essere utilizzato per ottenere la consistenza dei dati tra gli oggetti distribuiti.

Security Service. Esso permette ad un sistema ATC di definire una *repository* centralizzata per la gestione degli *account* degli utenti e di implementare meccanismi distribuiti di autenticazione.

Le *Common Facilities* definiscono dei *framework* definiti attraverso l'IDL che forniscono dei servizi che possono utilizzare direttamente le applicazioni. Nessuna delle *Common Facilities* fino ad oggi implementate si adattano al contesto dell'ATC.

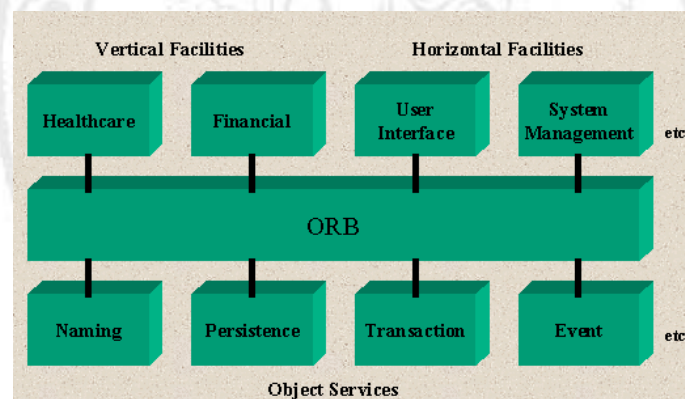


Figura 2.2 CORBA Services e Facilities.

Diverse implementazioni di CORBA soddisfano i requisiti dell' ATC, che sono stati definiti nel precedente paragrafo. Tuttavia si farà notare che si rendono necessarie ulteriori estensioni. Ad esempio un livello elevato di affidabilità può essere ottenuto introducendo meccanismi che permettono di tollerare i guasti. Pertanto CORBA offre meccanismi di tolleranza ai guasti per quei sistemi distribuiti che devono essere altamente affidabili e sempre disponibili.

In conclusione è possibile affermare che in scenari altamente critici, come quello del controllo del traffico aereo, si rende necessario la progettazione di middleware che oltre alle funzionalità base offra caratteristiche aggiuntive che permettano lo sviluppo applicazioni caratterizzate da un livello alto di affidabilità. Nel presente lavoro di tesi si farà riferimento ad un piattaforma middleware per sistemi *safety-critical*, CARDAMOM, che soddisfa molti dei requisiti emersi.

2.4 Middleware

Il middleware utilizzato è una piattaforma *Open Source* basata su CORBA, sviluppata in collaborazione da THALES e SELEX-SI ed orientata al contesto di sistemi *safety* e *mission-critical* tra cui sistemi per il controllo del traffico aereo (*ATC, Air Traffic Control*), sistemi navali e per la difesa. La struttura architettuale di base è organizzata a componenti –sia proprietari sia di terze parti (*COTS – Off the Shelf*) rendendo così semplice l' integrazione sia di componenti di business (che operano a livello funzionale) sia componenti tecnici che operanti a livello non funzionale. Detta integrazione tra componenti di diversa natura è inoltre resa possibile dalla conformità agli standard di riferimento più comuni:

- a livello di Business, UML (standard dell' *OMG – Object Management Group*) e XML (standard di *W3C - World Wide Web Consortium*);

- a livello di separazione dei compiti, CCM (*Corba Component Model*) che permette di separare la logica di business dell'applicazione dagli aspetti più propriamente tecnici.

L'infrastruttura offerta permette la configurazione, il *deployment* e l'esecuzione di sistemi distribuiti *near real-time* e *fault-tolerant*. Come per ogni sistema basato su CORBA esso è in grado di operare in scenari estremamente eterogenei, sia per ciò che concerne piattaforme hardware sia i linguaggi di programmazione. L'adozione del paradigma CCM, inoltre, permette agli sviluppatori di concentrarsi esclusivamente sulla logica di business dell'applicazione, rendendo possibile l'astrazione dei livelli più bassi, una più semplice attività di deployment nonché un incremento notevole della portabilità e del riuso del software. L'architettura di riferimento per cui il middleware usato è stato sviluppato è illustrato nella figura seguente, tuttavia esso è stato progettato per operare su diverse piattaforme hardware, tra i quali Windows e Unix, per diverse implementazioni di ORBs e per differenti linguaggi di programmazione tra i quali C++ e Java.

2.5 Servizi offerti

Dalla figura , che fornisce una visione generale dell' architettura, si evince come i servizi offerti dalla piattaforma siano di due tipi:

- ***basic services*** : ovvero System Management, ***Fault Tolerance*** e Repository.
- ***pluggable services***: ovvero Trace, Load Balancing, Recording, Time Management, Data Distribution, Transaction, Event Service, Persistence, Life Cycle.

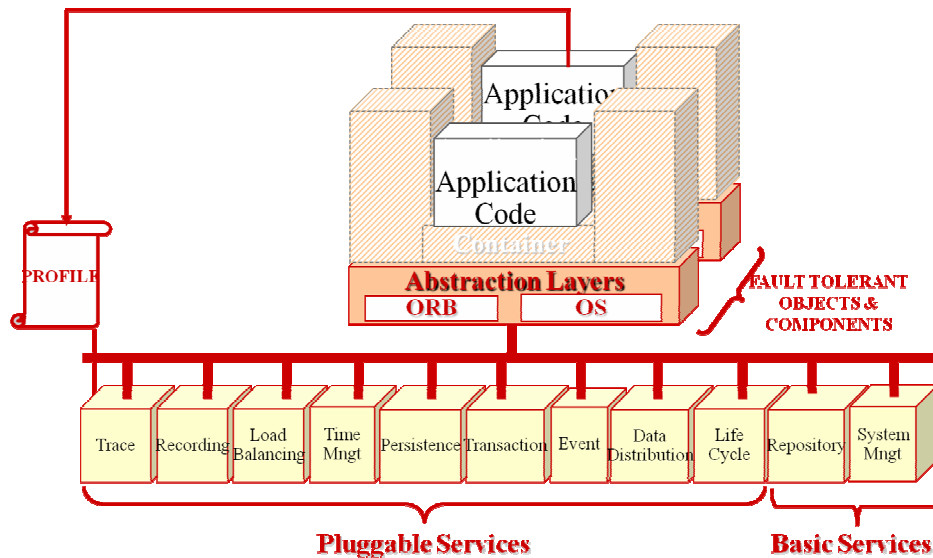


Figura 2-3 Architettura generale del middleware.

Il System Management è il servizio che supervisiona gli elementi che compongono un sistema, considerato come composto di nodi, applicazioni e processi. Con esso è possibile configurare un sistema, sia in fase di avvio sia in fase di esecuzione, oltre che avviare e arrestare l'intero sistema o parti di esso. Inoltre, esso consente il monitoraggio di processi e nodi, esegue la notifica automatica di specifici eventi e la notifica su richiesta dello stato del sistema.

Il servizio **Naming&Repository** consente di definire e ottenere oggetti attraverso i nomi, offrendo delle funzionalità simili a quelle offerte dallo standard COS Naming Service. Infatti l'accesso al servizio dei nomi può essere ottenuto attraverso l'interfaccia offerta dal COS Naming service, indipendentemente dall'implementazione dell'ORB.

Tutti i servizi si poggiano su un nucleo di base che va sotto il nome di *Middleware Core* (vedi figura 2-4) e che fornisce primitive per l'astrazione:

- a livello di Sistema Operativo, grazie al supporto di API quali ad esempio Thread, Mutex, Lock delle risorse;
- a livello di ORB, attraverso metodi per la creazione di POA con particolari politiche, e l'accesso a operazioni standard dell'Orb (ORB_Init, create_POA...).

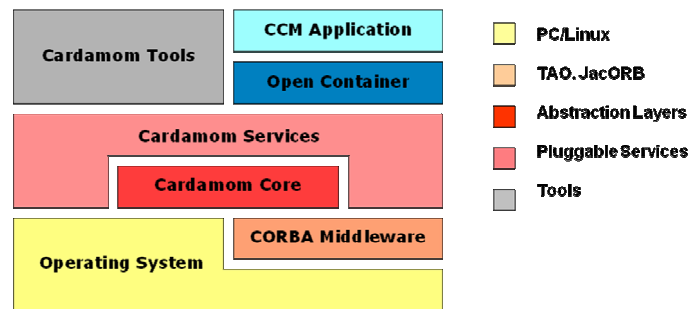


Figura 2-4: Architettura del Middleware Core.

Inoltre, *Middleware Core* fornisce anche :

- Un meccanismo di produzione di log in uno specifico formato.
- Un servizio per il *parsing* dei file XML.

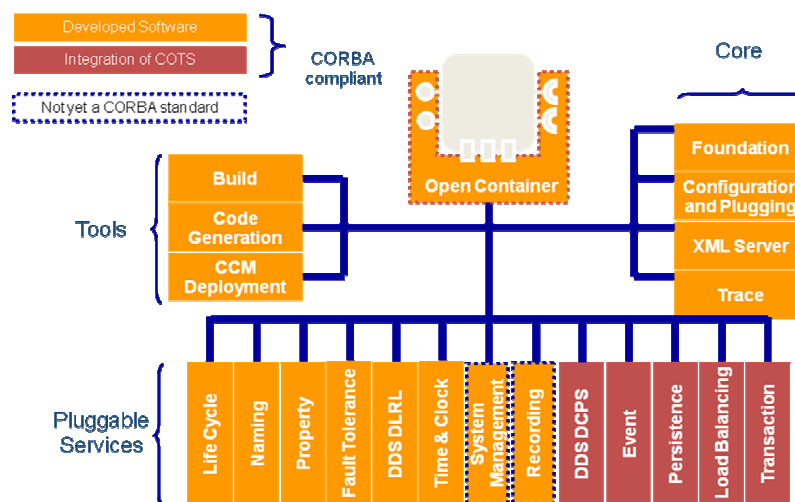


Figura 2-5: servizi, tools e core.

Parte integrante del middleware sono dei particolari tools, tra i quali si segnala il *Code Generator*, capace di produrre codice in linguaggio nativo (C++) a partire dai file di configurazione xml e dalle specifiche idl.

2.5.1 System Management

In questo paragrafo sarà presentata una panoramica del System Management. Esso permette di gestire il ciclo di vita sia dei processi di piattaforma (processi che includono solo software del middleware come ad esempio i servizi, sia quelli *Basic* che quelli *Pluggable*) sia dei processi utente (processi che includono software applicativo e librerie del middleware), permettendo così di evitare la gestione manuale degli stessi e di stabilire l'ordine di avvio dei processi.

I servizi offerti dal System Management permettono:

- di configurare il sistema, sia in fase di inizializzazione che a *run time*;
- di gestire il sistema, cioè permette l'avvio e lo stop dell'intero sistema o di sue parti, il *reboot* e lo *shutdown* dei nodi;
- di monitorare il sistema, con i relativi nodi e processi;
- di generare *report* e notifiche sullo stato del sistema, sia in seguito a richiesta sia in automatico al verificarsi di un particolare evento (si vedrà che queste proprietà vengano offerte rispettivamente dal Platform Admin e dal Platform Observer).

Per avere una visione chiara del System Management occorre definire alcuni concetti chiave. Innanzitutto, nella piattaforma utilizzata, un sistema gestito tramite System Management è composto da un certo numero di *hosts* e di applicazioni. Un' applicazione, a sua volta, non è altro che un insieme di processi che possono essere distribuiti su un certo numero di host. I processi possono essere di due tipi:

- **Processi *unmanaged* o *non collaborating***, i quali vengono soltanto creati dal System Management e possono essere arrestati stoppati soltanto arrestando il processo associato nel sistema operativo.
- **Processi *managed* o *collaborating***, per i quali il System Management gestisce l'intero ciclo di vita, cioè l'inizializzazione, l'avvio e lo stop. L'inizializzazione può prevedere anche diversi *step*.

La figura mostra la rete dei processi attraverso i quali il System Management può controllare un sistema.

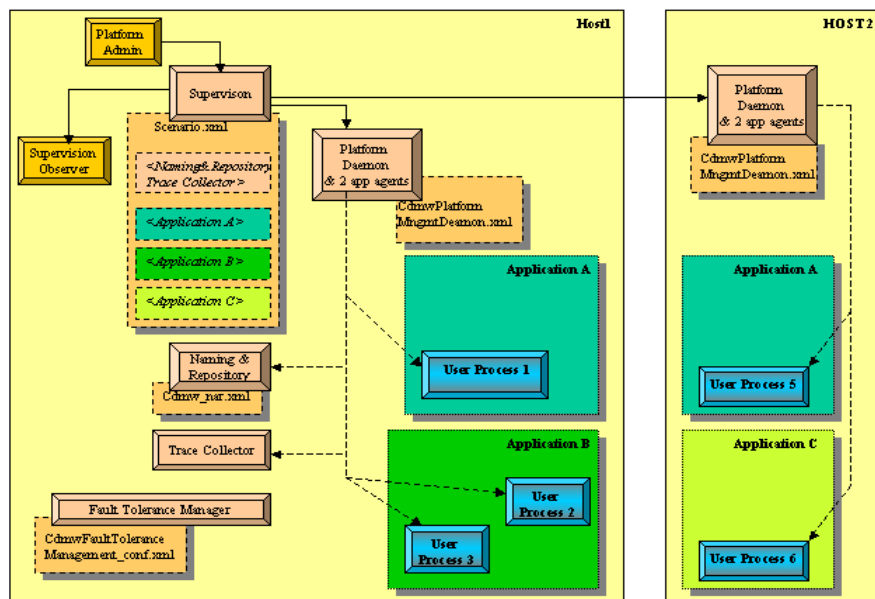


Figura 2-6 : processi del System Management.

In particolare si evidenziano:

- ✓ Il **Platform Daemon**. Esiste un solo processo di questo tipo per ciascun nodo; esso si occupa dell'avvio, dello stop e del monitoraggio degli Application Agent in esecuzione sullo stesso nodo.

- ✓ **L'Application Agent.** Si occupa dell'avvio, dello stop e del monitoraggio dei processi dell'applicazione che esso gestisce e che sono in esecuzione su quel nodo.
- ✓ **Il Platform Supervision Server.** Esiste un unico processo di questo tipo nell'intero sistema, che supervisiona le applicazioni e gli host che lo compongono. Attraverso opportune interfacce IDL, esso permette ai *Supervision Clients* lo start e lo stop di applicazioni e processi che esso gestisce, lo *shutdown* e il *restart* degli *host*. Esso è a conoscenza dello stato delle applicazioni, dei processi e degli *host*. A chi sviluppa è data la possibilità di controllare il *Supervision*.
- ✓ **Supervision Clients.** Attraverso le interfacce IDL esportate dal *Supervision Server*, esso controlla l'intero sistema. Offre due *Supervision Client* di *default*:
 - ✓ **Il Platform Admin.** È un HMI (*Human Machine Interface*) a linea di comando che accetta comandi per il controllo del sistema (cambiamenti della configurazione, avvio e stop di applicazioni etc.).
 - ✓ **Il Platform Observer.** Esso genera allarmi e segnala i cambiamenti di stato di *host*, applicazioni e processi che vengono inviati dal *Platform Supervision*.

Un processo utente gestito dal System Management deve essere in grado di processare le richieste per l'inizializzazione, il run e lo stop inviate dalla piattaforma di gestione e, a questa, deve notificare i cambiamenti di stato ed eventuali rilevamenti di errori. Per realizzare questo meccanismo di comunicazione, la piattaforma mette a disposizione una particolare libreria che include dei servants CORBA che elaborano le richieste inviate dalla piattaforma di gestione, sollevando i developers da questo ulteriore compito. La comunicazione tra servants e l'applicazione fa uso del meccanismo comunicazione asincrona basato su callback, che, in uno scenario client-server, evita che il flusso del programma client si blocchi in attesa della risposta del server. Infatti il server una volta terminata l'elaborazione può restituire il risultato al client,

richiamando un' operazione sull'oggetto client il cui riferimento gli è stato fornito insieme alla richiesta.

2.6 Ciclo di sviluppo: il “Modello a V”

Il ciclo di sviluppo della piattaforma middleware oggetto del nostro lavoro segue un particolare modello di sviluppo denominato “Modello a V”.

Secondo tale modello, mostrato in figura , le specifiche del software vengono definite seguendo un particolare gerarchia.

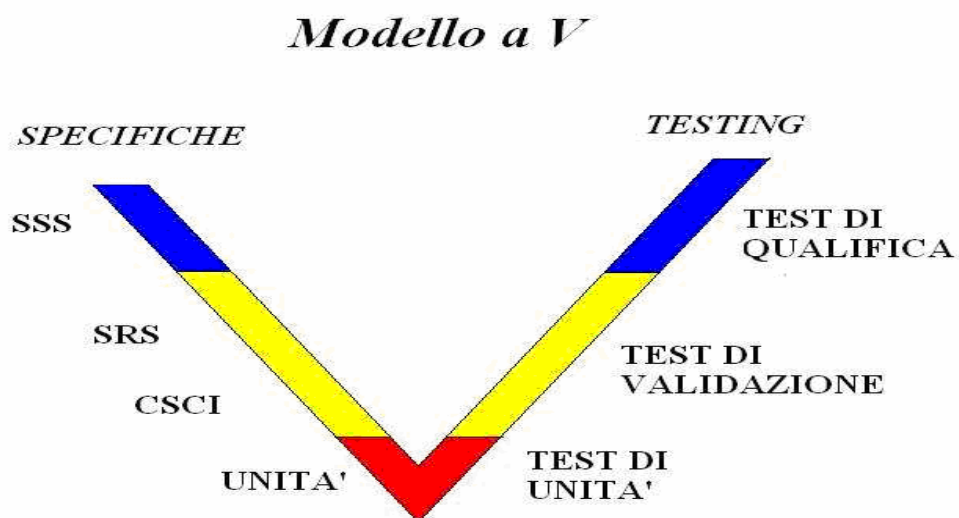


Figura 2-7 :Modello a V

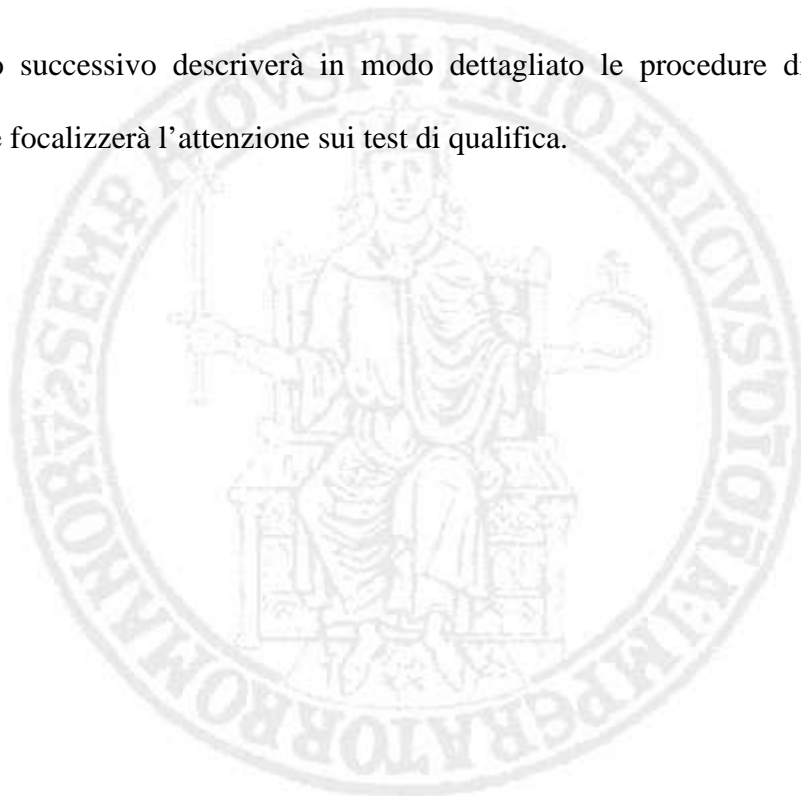
Ad un livello più alto vi sono le specifiche SSS (System Segment Specifications) cui seguono le SRS (Software Requirements Specifications); il soddisfacimento di ogni singola specifica SRS consente il soddisfacimento di un intera specifica SSS. Ogni specifica SRS, inoltre, è relativa ad un singolo CSCI (Computer Software Configuration Item). Ad un livello più basso ogni singolo CSCI si compone di diverse unità: il

soddisfacimento delle specifiche per ogni singola unità consente il soddisfacimento di un intero CSCI.

Dopo le specifiche di unità, risalendo il grafico come mostrato in figura, comincia la fase di testing. Il livello più basso riguarda, ovviamente, i test di unità; vi sono poi i test di validazione che effettuano il testing di un insieme di unità, viste nel loro complesso come singolo modulo CSCI. Il successo dei test di validazione comporta, dunque, il corretto soddisfacimento delle specifiche SRS. A sua volta i test di qualifica sono atti a verificare il funzionamento di un insieme di moduli CSCI, il cui successo comporta il corretto soddisfacimento delle specifiche di più alto livello SSS.

In tale modello, inoltre, l'insuccesso del testing di qualifica comporta la revisione del testing dei relativi moduli CSCI; allo stesso modo l'insuccesso di un singolo test di validazione comporta la revisione del testing delle relative unità.

Il capitolo successivo descriverà in modo dettagliato le procedure di testing ed in particolare focalizzerà l'attenzione sui test di qualifica.



Capitolo 3

Testing della piattaforma middleware

3.1 Procedure di testing

Il testing software è un processo utilizzato per valutare la qualità dell'applicazione sviluppata. Solitamente il concetto di qualità di un'applicazione è profondamente legato a parametri come la correttezza, completezza, sicurezza, affidabilità, efficienza, portabilità ed usabilità.

Il testing è quell'attività di "esercizio" del software tesa all'individuazione dei malfunzionamenti prima della messa in esercizio e, con riferimento alla tesi di Dijkstra, non può dimostrare l'assenza di difetti, ma ne può solo dimostrare la presenza. Il testing, quindi, include ma non è strettamente limitato all'esecuzione di un programma con l'intento di trovarne malfunzionamenti.

Risulta evidente come il testing costituisca una delle fasi fondamentali nel processo di sviluppo software fornendo importanti informazioni riguardo l'adempimento di particolari specifiche di sistema.

Per quanto riguarda la piattaforma middleware in esame, la definizione stessa di

una procedura di testing risulta particolarmente complessa in quanto tale risulta l'architettura stessa del sistema e le specifiche a cui esso si riferisce richiedono il rispetto di elevati standard che fanno della fase di testing della piattaforma un nodo cruciale dell'intero ciclo di sviluppo.

Durante la fase di testing è strettamente indispensabile tenere in conto dell'ambiente in cui il middleware si troverà poi ad essere eseguito: è necessario ricreare un ambiente virtuale (Figura 3.1) che emuli perfettamente un ambiente distribuito in cui ognuno dei nodi soddisfi i seguenti requisiti hardware e software:

- Le macchine devono essere equipaggiate con sistema operativo Linux, compilatore C++, Java
- Lo switch di rete deve essere a 100 Mbit/s

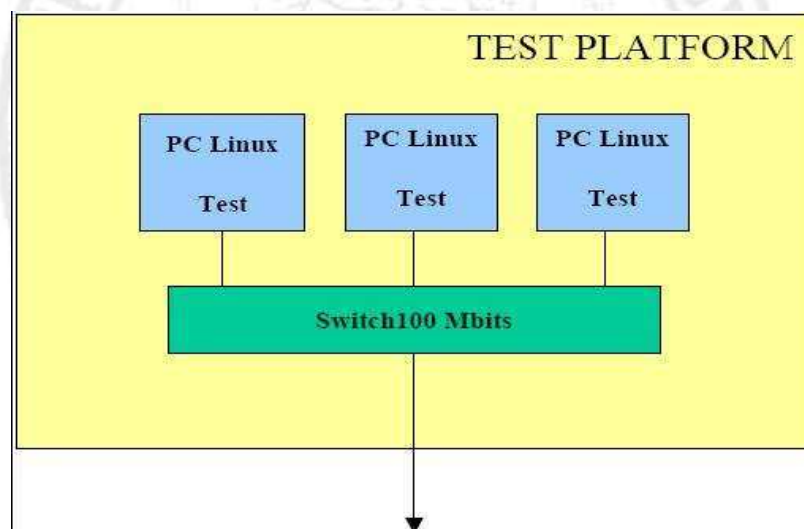


Figura 3.1 Ambiente virtuale

La configurazione minimale prevede inoltre l'utilizzo di almeno 2 host, di una rete locale per la comunicazione tra i nodi e di una macchina dedicata all'analisi off-line dei risultati. Sulla base di questa configurazione fisica si può creare una qualsiasi configurazione logica, facendo variare il numero e il ruolo dei client, dei server e delle altre entità in gioco.

Per poter effettuare il testing è necessario eseguire le seguenti operazioni:

- Installare i test di qualifica
- Definire un profilo

Un profilo non è altro che la descrizione di una specifica parte dell'ambiente di test, ed ha lo scopo di disaccoppiare il test dall'ambiente operativo in cui sarà eseguito.

Così facendo la descrizione del test si focalizza sullo scenario di esecuzione e non sulla piattaforma, sul sistema operativo, sul particolare ORB, etc. Risulta, quindi, possibile eseguire test diversi sullo stesso profilo, e il medesimo test su profili diversi.

E' possibile individuare tre categorie di test:

- *Test di Unità*
- *Test di Validazione*
- *Test di Qualifica*

Procediamo ad una breve descrizione di tali categorie.

3.1.1 Test di unità

Affrontano ciascun componente separatamente, assicurando il suo corretto funzionamento come unità a sé stante. Test di questo tipo fanno largo uso di tecniche di testing di tipo white-box. In questa fase sono eseguiti i cammini specifici nella struttura di controllo della singola unità per assicurare la copertura completa e il rilevamento del maggior numero di errori. In seguito i moduli verranno integrati tra loro e testati nelle loro interazioni con le

altre unità (test di integrazione);

3.1.2 Test di validazione

Questi test hanno lo scopo di verificare che un CSCI (macrocomponente) soddisfi completamente i requisiti SRS stabiliti nel relativo documento delle specifiche di progetto.

Nei fatti i test di validazione vengono suddivisi per CSCI, cioè test del System Management, dell'Event Service, ed il singolo test verifica il soddisfacimento di parte delle SRS.

Ad esempio il test V-SMG-003 è un test relativo al System Management, che deve verificare la copertura di un certo numero di specifiche così come riportato in tabella.

REQ-SRS-SMG-191	REQ-SRS-SMG-201	REQ-SRS-SMG-205	REQ-SRS-SMG-250	REQ-SRS-SMG-270
REQ-SRS-SMG-272	REQ-SRS-SMG-274	REQ-SRS-SMG-285	REQ-SRS-SMG-298	REQ-SRS-SMG-300
REQ-SRS-SMG-330	REQ-SRS-SMG-340			

V-SMG-003

3.1.3 Test di qualifica

Sono una serie di test formalmente definiti, tramite i quali è possibile stimare le performance di un singolo componente o dell'intero sistema, in termini di affidabilità ed altri parametri funzionali e non funzionali.

A differenza di quelli precedenti vanno a sollecitare più moduli CSCI ed a verificare la loro interoperabilità. Così come accennato nel paragrafo 2.6, un test di qualifica è eseguito per verificare il soddisfacimento delle specifiche di alto livello SSS. Quest'ultime vanno considerate come requisiti di prodotto che includono sia aspetti funzionali che non funzionali, interfaccia esterna e vincoli

di realizzazione.

Nei fatti un test di qualifica deve verificare la copertura di un certo numero di specifiche SSS e per fare questo può sollecitare uno o più moduli CSCI. In tabella è riportato il caso del test Q-001.

REQ-SSS-CPB-0010	REQ-SSS-CPB-0390	REQ-SSS-INT-0011	REQ-SSS-CPB-0012
------------------	------------------	------------------	------------------

Q-001

3.2 Analisi dei Test di qualifica

I *Test di Qualifica* comprendono diversi tipi di test, tali da raggiungere una copertura totale di tutti i requisiti:

- ***Inspection***, che prevede un'ispezione del codice e della documentazione fornita dal Middleware;
- ***Analysis***, che effettua l'analisi dei dati collezionati;
- ***Demonstration***, che effettua la verifica delle operazioni svolte e fornisce risultati qualitativi;
- ***Test***, che prevede la verifica delle operazioni e dei requisiti funzionali con lo scopo di generare dati da sottoporre in ingresso e collezionare per ulteriori studi.

I test di qualifica, inoltre, trattano aspetti funzionali per verificare se il prodotto soddisfa i requisiti funzionali e non-funzionali del CSCI, come le performance e il consumo di risorse. Essi operano su tre distinti livelli:

- *Single Unit*, in cui si testa la singola unità
- *CSCI*, in cui si verificano i requisiti di tutto il *CSCI*
- *Product*, che testa il *CSCI* integrato con altro software

Vediamo adesso nel dettaglio la procedura di esecuzione di un test di qualifica.

3.2.1 Procedura di lancio

La procedura di esecuzione di un test di qualifica si divide in due fasi: la prima è di preparazione e di avvio del test, ed è a cura di un operatore; la seconda, invece, è quella di esecuzione vera e propria, ed è a cura della macchina su cui il test è eseguito.

La fase di preparazione e di avvio di un test può variare anche nell'ambito dei diversi scenari di un dato test, difatti esistono alcuni scenari di test con procedure di esecuzione totalmente diverse e talvolta uniche. Tale fase, inoltre, è un'operazione lunga e molto complessa da eseguire: un operatore esperto può impiegare anche 12 minuti per l'esecuzione del primo test, 5 minuti per i successivi in quanto alcune istruzioni devono essere eseguite solo quando viene avviato il primo test e non occorre eseguirle nuovamente per i test successivi.

Riportiamo di seguito (Figura) il "Sequence Diagram" della fase di preparazione e di avvio di un test di qualifica di esempio, al fine di mostrare la sua complessità.

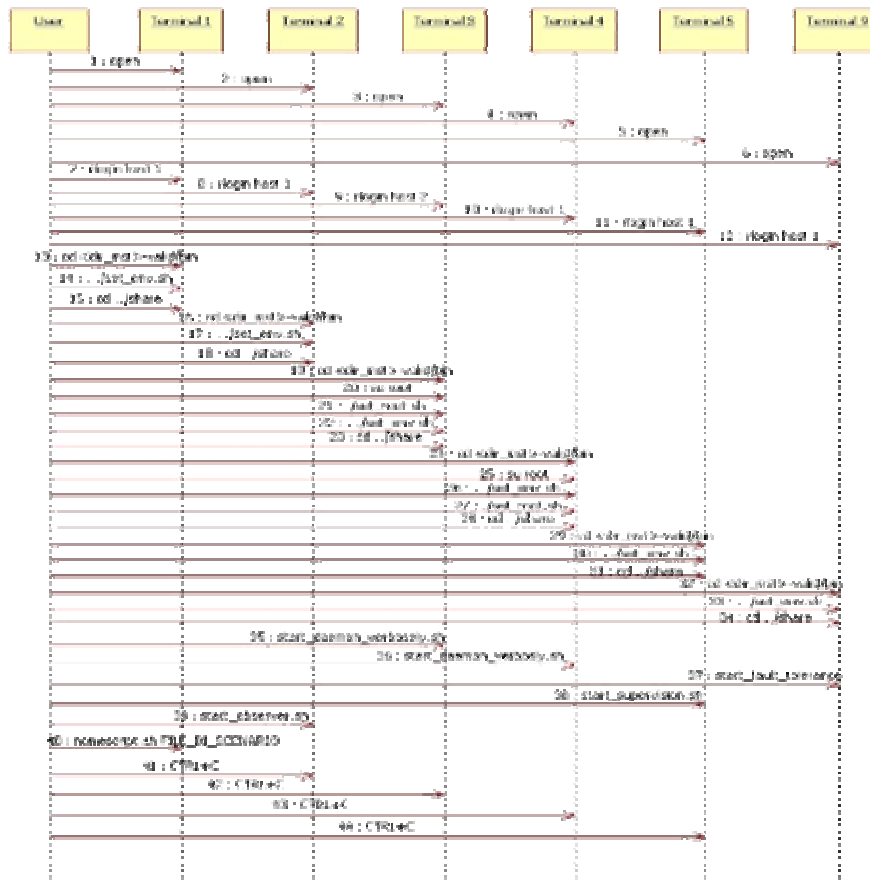


Figura 3-2 Esecuzione test di qualifica

La procedura di lancio del 50 % dei test possiede una procedura di avvio e preparazione test uguale. Tale procedura può essere suddivisa in 5 fasi sequenziali :

1. preparazione dell'ambiente (da eseguire solo per il primo test);
2. configurazione dell'ambiente (solo per il primo test);
3. avvio dei demoni (per ogni test);
4. esecuzione del test (per ogni test);
5. uccisione demoni e chiusura terminali(per ogni test).

Presentiamo in dettaglio ciascuna delle fasi appena elencate:

1. Preparazione ambiente.

La preparazione dell'ambiente prevede la presenza di due host e di sei terminali di lavoro (TERM1, TERM2, TERM3, TERM4, TERM5, TERM9) connessi agli

hosts nel seguente modo:

- TERM1 connesso a HOST1
- TERM2 connesso a HOST1
- TERM3 connesso a HOST2
- TERM4 connesso a HOST1
- TERM5 connesso a HOST1
- TERM9 connesso a HOST1

Se non si è già connessi ai relativi host è necessario farlo digitando il seguente comando:

```
rlogin  
HOST
```

2. Configurazione dell'ambiente.

Sul primo terminale di lavoro (TERM1) occorre effettuare le seguenti operazioni:

Impostare la directory di lavoro corrente:

```
cd <directory di installazione della piattaforma >-qualif/bin
```

Impostare le variabili di ambiente richieste digitando il seguente comando:

```
../set_env.sh
```

Impostare la directory dove risiedono i file di configurazione:

```
cd ../share
```

Sul secondo terminale di lavoro (TERM2) occorre effettuare le seguenti operazioni:

Impostare la directory di lavoro corrente:

```
cd <directory di installazione della piattaforma>-valid/bin
```

Impostare le variabili di ambiente richieste digitando il seguente comando:

```
./set_env.sh
```

Impostare la directory dove risiedono i file di configurazione:

```
cd ../share
```

Sul terzo terminale di lavoro (TERM3) occorre effettuare le seguenti operazioni:

Impostare la directory di lavoro corrente:

```
cd <directory di installazione della piattaforma>-qualif/bin
```

Settare l'account di root digitando il seguente comando:

```
su root
```

```
(password  
relativa)
```

Impostare i diritti di accesso digitando il seguente comando:

```
./set_root.sh
```

Impostare le variabili di ambiente richieste digitando il seguente comando:

```
./set_env.sh
```

Impostare la directory dove risiedono i file di configurazione:

```
cd ../share
```

Sul quarto terminale di lavoro (TERM4) occorre effettuare le seguenti operazioni:

Impostare la directory di lavoro corrente:

cd <directory di installazione della piattaforma>-qualif/bin

Settare l'account di root digitando il seguente comando:

su root

*(password
relativa)*

Impostare i diritti di accesso digitando il seguente comando:

. ./set_root.sh

Impostare le variabili di ambiente richieste digitando il seguente comando:

. ./set_env.sh

Impostare la directory dove risiedono i file di configurazione:

cd ../share

Sul quinto terminale di lavoro(TERM5) occorre effettuare le seguenti operazioni:

Impostare la directory di lavoro corrente:

cd <directory di installazione della piattaforma>-qualif/bin

Impostare le variabili di ambiente richieste digitando il seguente comando:

. ./set_env.sh

Impostare la directory dove risiedono i file di configurazione:

cd ../share

Sul sesto terminale di lavoro(TERM9) occorre effettuare le seguenti operazioni:

Impostare la directory di lavoro corrente:

cd <directory di installazione della piattaforma>-qualif/bin

Impostare le variabili di ambiente richieste digitando il seguente comando:

. ./set_env.sh

Impostare la directory dove risiedono i file di configurazione:

```
cd ../share
```

3. Avvio dei demoni.

Per l'esecuzione dei test occorre avviare sui terminali di lavoro dei demoni opportuni mediante le seguenti operazioni:

Sul TERM3 avviare il demone di gestione del sistema se non già avviato:

```
log.sh start_daeomon_verbosely.sh
```

Sul TERM4 (connesso all'host remoto) avviare il demone di gestione del sistema se non già avviato:

```
log.sh start_daeomon_verbosely.sh
```

Sul TERM9, solo nei test in cui è richiesta la presenza del fault tolerance manager, si deve eseguire il seguente comando:

```
log.sh cdmw_ft_manager -  
CdmwXMLFile=CdmwFaultToleranceManager_ccmft_conf.xml-  
groupConf=CdmwFTSystemMngtGroupCreator_conf.xml
```

Sul TERM5 avviare il server di supervisione:

```
log.sh start_supervision.sh
```

Sul TERM2 avviare l'osservatore di supervisione:

```
log.sh start_observer.sh
```

4. Esecuzione del test

Per avviare l'esecuzione del test occorre digitare sul TERM1 il seguente comando:

nomescript.sh FILE_DI_SCENARIO

dove il “nomescript.sh” varia a seconda del test effettuato ed il file di scenario è del tipo “Q-XXX-YYY” in cui XXX è il numero del test.

5. Uccisione demoni e chiusura terminali

Per una corretta terminazione dei demoni avviati occorre eseguire sul TERM2, TERM3, TERM4, TERM5 il comando “CTRL+C”.

Occorre poi chiudere i terminali manualmente.

3.2.2 Procedura di valutazione

La procedura di valutazione dei risultati può variare per ogni singolo test; esistono, tuttavia dei criteri di carattere generale descritti di seguito(vedi paragrafo 6.3).

Se il test è andato a buon fine viene visualizzato sul TERM1 il seguente messaggio:

The qualification scenario is OK

In caso contrario viene visualizzato:

FAILED

Vengono inoltre prodotti due file di log nella directory “Q-XXX-results”, dove XXX è il numero del file di scenario. I file di log prodotti sono: “scenario.result” ed “-execution-<HOST>.output” dove <HOST> è il nome dell' host su cui è stato eseguito il test. Il primo contiene l'esito del test, mentre il secondo contiene il risultato di tutte le singole operazioni svolte dall'host durante il test.

3.2.3 Problemi riscontrati

Dall'analisi effettuata in precedenza, è abbastanza immediato derivare quali sono le difficoltà principali dovute all'esecuzione di un singolo test:

- procedura complessa e macchinosa: l'utente deve eseguire un numero sufficientemente elevato di operazioni prima di poter effettivamente lanciare un test; un utente esperto può impiegare per l'avvio di un test anche circa 11 minuti (primo test) o 4 minuti (test successivi);
- l'avvio dei demoni deve rispettare un certo ordine;
- durante l'esecuzione di uno specifico test, l'utente è costretto ad interagire continuamente con la macchina per garantire la terminazione del test.
- L'esecuzione dell'insieme di test richiede un tempo notevole

L'obiettivo del seguente lavoro di tesi è proprio quello di superare tali difficoltà, fornendo all'utente uno strumento semplice e rapido per l'esecuzione automatica dei test di qualifica.

Capitolo 4

Dall'analisi alla progettazione del TOOL

In questo capitolo vengono descritte le fasi di analisi e progettazione del tool.

4.1 Obiettivi del tool

L'esecuzione di un singolo test di qualifica presenta le seguenti problematiche:

1. risulta un'operazione complessa e macchinosa, in quanto l'operatore deve eseguire una lunga lista di comandi prima di poter effettivamente lanciare un test;
2. è un'operazione che richiede un tempo elevato a causa della quantità di comandi da eseguire; un operatore esperto può impiegare circa 11 minuti per lanciare il primo test e 4 minuti per i successivi;
3. è richiesta una continua interazione utente-macchina: l'utente deve interagire continuamente con la macchina affinché il test possa proseguire e terminare correttamente;
4. è necessario eseguire i comandi con un certo ordine pena l'invalidazione della procedura di lancio.
5. scarsa leggibilità e comprensione dei risultati: vi è una sovrapposizione dei

terminali di lavoro aperti da uno specifico caso di test, che rende poco chiara la comprensione di quali sono le operazioni svolte dalla macchina durante l'esecuzione di uno specifico test.

La realizzazione di un tool per l'automatizzazione dei test di qualifica nasce principalmente con lo scopo di risolvere i problemi appena elencati; Di seguito vengono descritte le fasi di analisi e progettazione, realizzate seguendo le tecniche proprie dell'ingegneria del software.

4.2 Analisi e Specifica dei Requisiti

L'attività di analisi e specifica dei requisiti ha lo scopo di definire COSA il sistema dovrà fare senza descrivere il COME.

Il tool per la verifica software di un sistema distribuito nasce con lo scopo di risolvere le problematiche inerenti l'esecuzione dei test di qualifica.

I requisiti che il tool dovrà possedere sono i seguenti :

- capacità di eseguire in maniera automatica tutta la fase di preparazione e avvio dei test di qualifica;
- possibilità di utilizzare due modalità di funzionamento, una manuale per l'esecuzione di un singolo test e una automatica per l'esecuzione di un intero set di test di qualifica;
- eliminazione dell'interazione utente-macchina durante l'esecuzione di uno specifico test;
- riempimento dei menu dei test tramite file di configurazione;
- posizionamento migliore e chiusura dei terminali a valle dell'esecuzione dei test (modalità manuale) e presenza di una repository dei risultati

(modalità automatica).

Passiamo ora ad individuare i casi d'uso del sistema da progettare.

4.2.1 Individuazione dei casi d'uso

I casi d'uso mostrano come un'entità esterna al sistema (l'utente) può interagire col sistema stesso; per il sistema in esame, i casi d'uso possono essere specificati ricorrendo al seguente diagramma :

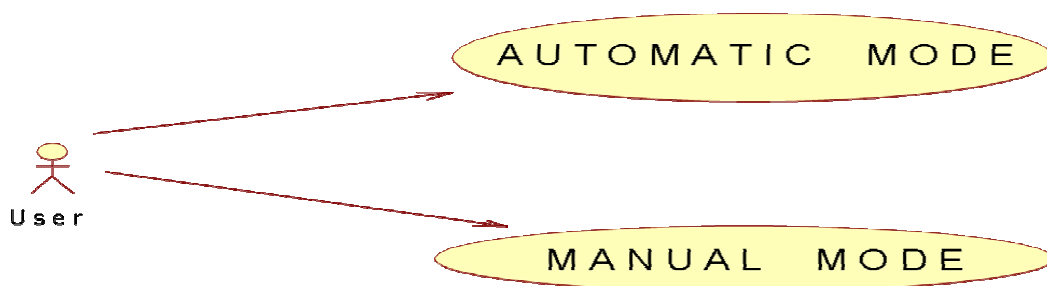


Figura 4.1 Diagramma dei casi d'uso

Come si vede dal diagramma dei casi d'uso, bisogna consentire all'utente di interagire col sistema utilizzando due modalità di funzionamento:

- manuale per l'esecuzione di un singolo test
- automatica per l'esecuzione di una lista di test.

Ciascuna delle due modalità include altre operazioni necessarie al soddisfacimento dei compiti richiesti.

4.2.2 Specifica dei casi d'uso

La specifica dei casi d'uso fornisce una descrizione dettagliata di ciascun caso d'uso presentato nel precedente diagramma.

Considereremo i casi d'uso "AutomaticMode" e "ManualMode".

4.2.2.1 Caso d'uso "AutomaticMode"

Lo scenario di esecuzione del caso d'uso "AutomaticMode" è il

seguinte: **Nome:** AutomaticMode

Iniziatore: Utente generico

Obiettivo: Esecuzione automatica di una lista di test di qualifica

Scenario principale:

1. L'utente seleziona la modalità automatica
2. Vengono caricati da un file di configurazione (formato xml) i parametri di default necessari all'esecuzione di una lista di test
3. L'utente può modificare i parametri di default selezionando un altro file da cui prelevare i test da eseguire
4. L'utente avvia il test
5. Viene effettuato un controllo sui parametri inseriti
6. Vengono avviati in sequenza i test contenuti nel file di testo specificato
7. Viene eseguita la procedura automatica di preparazione e avvio di ciascun test
8. Terminati i test, viene presentata una schermata con i risultati dei test
9. L'utente seleziona il particolare test
10. Vengono visualizzati i dettagli del particolare test selezionato

Extends:

5. parametri di input errati
- 5.a Viene richiesto all'utente di reinserire i parametri errati

4.2.2.2 Caso d'uso "ManualMode"

Lo scenario di esecuzione del caso d'uso "ManualMode è il

seguito: **Nome:** ManualMode

Iniziatore: Utente generico

Obiettivo: Esecuzione automatica di un singolo test di qualifica

Scenario principale:

1. L'utente seleziona la modalità manuale
2. Vengono caricati da un file di configurazione (formato xml) i parametri di default necessari all'esecuzione del singolo test
3. L'utente seleziona il particolare test da eseguire
4. L'utente avvia il test
5. Viene effettuato un controllo sui parametri inseriti
6. Viene eseguita la procedura automatica di preparazione e avvio del test
7. Vengono aperti i terminali necessari in posizione corretta
8. Sul primo terminale viene visualizzato il risultato del test
9. L'utente preme un tasto dopo aver osservato il risultato del test
10. Si chiudono tutti i terminali aperti e vengono uccisi i demoni avviati

Extends:

5. parametri di input errati
- 5.a Viene richiesto all'utente di reinserire i parametri errati

4.2.3 Sequence Diagram per i casi d'uso

Una descrizione dettagliata e compatta delle operazioni che ciascuna delle due modalità dovrà essere in grado di svolgere può essere fornita utilizzando dei Sequence Diagram.

Per quanto concerne la **modalità automatica**, è possibile schematizzare le azioni che

dovranno essere eseguite nel seguente modo :

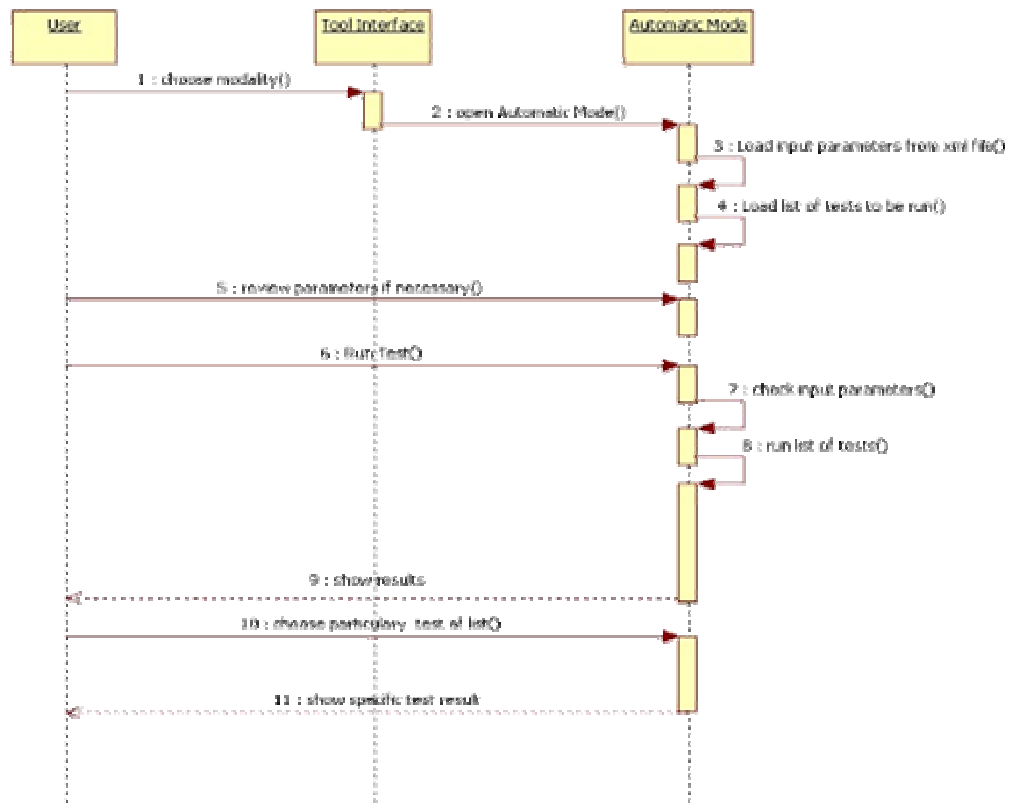


Figura 4.2 Sequence Diagram Automatic Mode

Con la presenza dell'interfaccia grafica, l'utente dovrà soltanto selezionare la particolare modalità e avviare il test. Tutta la fase di preparazione e di gestione dei test dovrà essere eseguita automaticamente dal tool che solleverà quindi l'utente dall'onere di interagire con la macchina durante l'esecuzione di uno specifico test. Tale modalità dovrà anche fornire una funzionalità che consenta all'utente di eseguire un'intera lista di test.

Per quanto riguarda invece la **modalità manuale**, è possibile utilizzare il seguente schema per comprendere quali saranno le funzionalità che essa dovrà fornire all'utente :

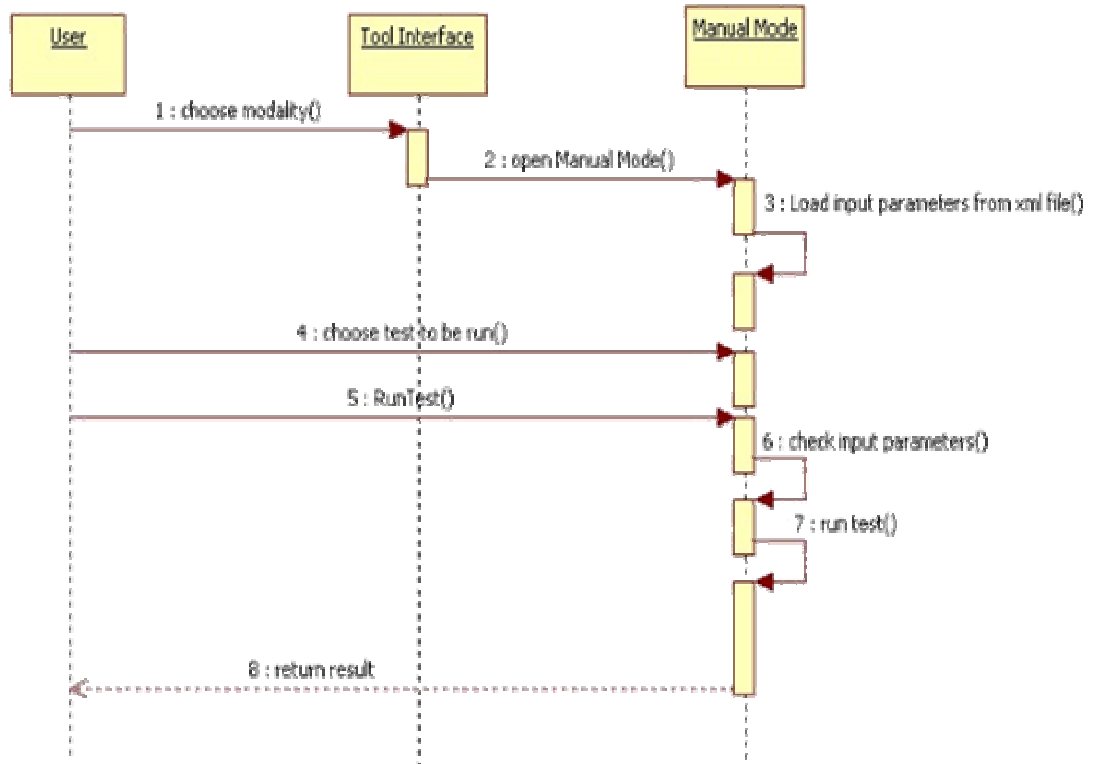


Figura 4.3 Sequence Diagram Manual Mode

Anche la modalità manuale dovrà fornire gli stessi vantaggi visti per la modalità automatica, con l'unica differenza che essa riguarderà soltanto l'esecuzione di un singolo test e non di un'intera lista di test.

4.3 Progettazione

Dall'analisi e dalla specifica dei casi d'uso, osserviamo che l'obiettivo principale è quello di automatizzare tutte le operazioni necessarie all'esecuzione dei test. Una prima analisi porterebbe quindi a pensare di organizzare il tool in due macrocomponenti: l'interfaccia e il core dell'applicazione.

Si potrebbe quindi definire il seguente diagramma delle classi :

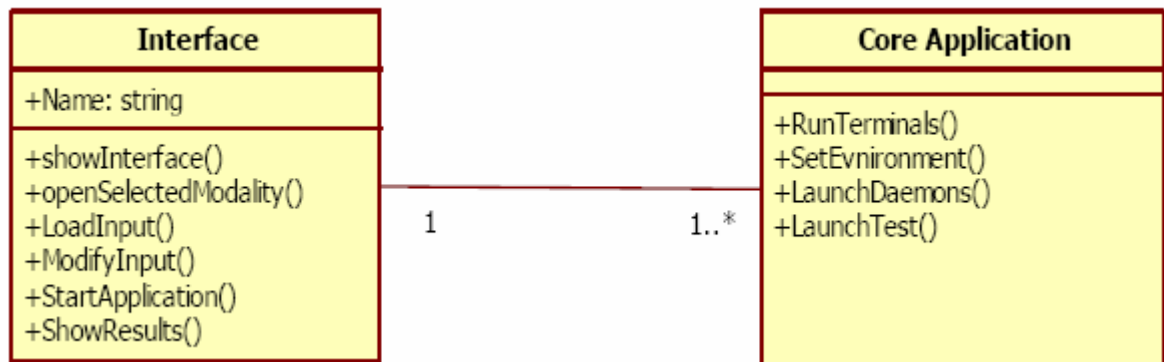


Figura 4.4 Organizzazione progettuale ad alto livello

L'**interfaccia** deve fornire metodi che consentano all'utente di :

- Selezionare la modalità di avviamento del tool (automatica o manuale);
- Caricare automaticamente gli input necessari;
- Modificare ove necessario gli input di default;
- Selezionare il test o l'elenco di test da mandare in esecuzione;
- Avviare i test;
- Visualizzare i risultati dell'esecuzione dei test.

Il **core** ha il compito di eseguire tutte le operazioni di basso livello, ossia le operazioni necessarie alle fasi di preparazione e avvio dei test e alla copia dei file necessari sull'host remoto. L'utente non si interfacerà mai con il core dell'applicazione, ossia non utilizzerà mai i metodi messi a disposizione da quest'ultimo ma sarà l'interfaccia a richiamarli in modo da facilitare i compiti dell'utente e ridurre la probabilità di errore.

Vediamo in dettaglio ciascuno dei due macrocomponenti.

4.3.1 Graphical User Interface

Come detto in precedenza, l'interfaccia ha lo scopo di eseguire in maniera automatica tutte le operazioni di preparazione e avvio dei test in modo che l'utente non debba più eseguire manualmente tali comandi. Quindi la GUI ha il compito di interfacciarsi con l'utente; qualsiasi operazione l'utente richieda all'interfaccia verrà demandata al Core il quale si occupa delle fasi di configurazione delle macchine e di avvio dei test.

Ovviamente, eventuali errori in fase di gestione errata degli input dovranno essere segnalati all'utente attraverso opportuni messaggi.

Per capire in che modo la GUI si interfaccia con il Core dell'applicazione osserviamo il seguente diagramma :



Figura 4.5 *Interazione GUI-CORE*

Quindi la GUI richiederà al Core di eseguire un particolare test; tutte le operazioni necessarie alla preparazione e avvio del test verranno svolte internamente dal Core. Abbiamo visto che queste operazioni, se svolte dall'utente, richiedono un tempo sufficientemente elevato; utilizzando invece un'interfaccia grafica, l'utente dovrà soltanto avviare l'esecuzione del test con un apposito comando fornito dall'interfaccia, senza preoccuparsi di dover eseguire ogni volta tutte le operazioni necessarie al funzionamento del test, operazioni che come detto verranno eseguite internamente dal Core dell'applicazione su ordine

dell'interfaccia

Per comprendere le interazioni User-GUI-Core è possibile ricorrere al seguente Sequence Diagram :

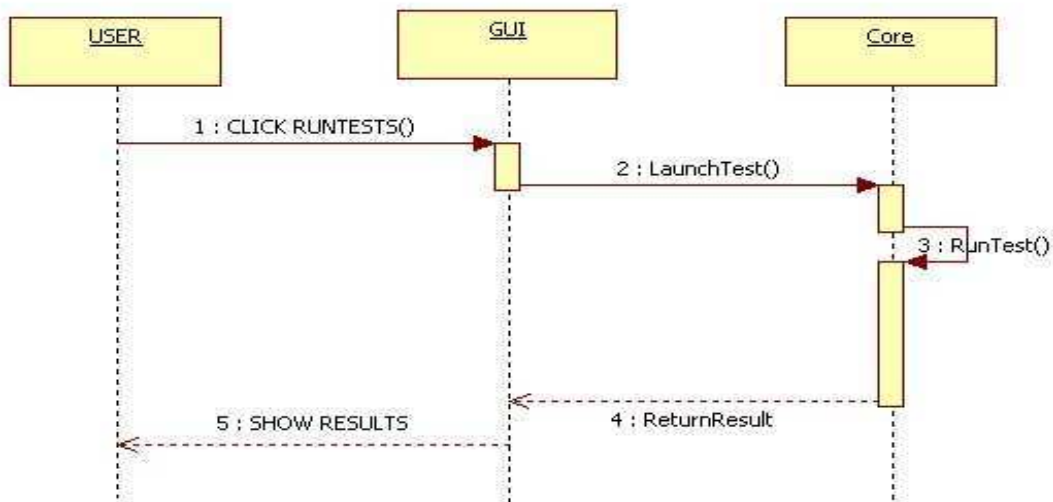


Figura 4.6 Interazioni User-GUI-Core

Come si può vedere dal diagramma, l'utente interagirà solo con l'interfaccia grafica, mentre quest'ultima richiederà direttamente al Core dell'applicazione di effettuare tutte le operazioni necessarie all'esecuzione del test.

Scendendo ad un maggior livello di dettaglio, l'interfaccia dovrà prevedere due modalità di funzionamento:

- **Manuale**, per l'esecuzione di un singolo test di qualifica;
- **Automatica**, per l'esecuzione di un set di test di qualifica.

Dunque è opportuno utilizzare una nuova classe che indichi la modalità di avviamento del tool da realizzare.

Il diagramma delle classi può essere particolarizzato come segue :

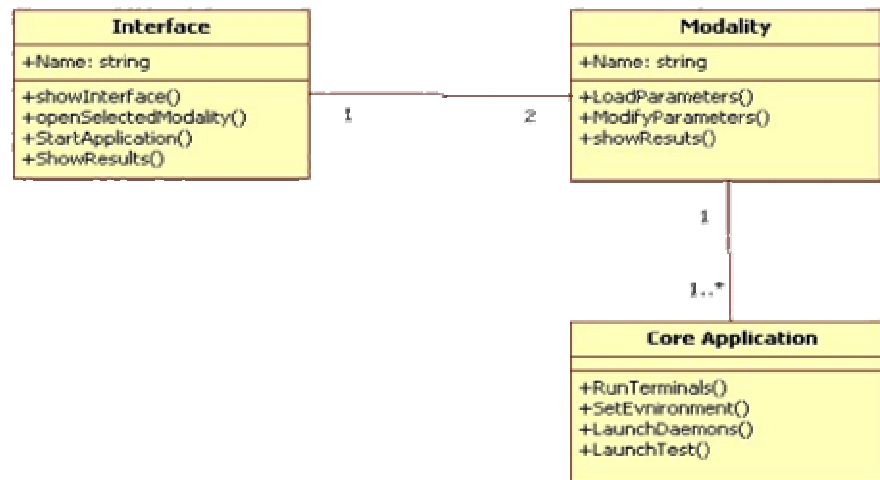


Figura 4.7 Class Diagram per la GUI

Sarà quindi la classe **Modality** ad interfacciarsi con il **Core** dell'applicazione, richiedendo particolari servizi in base alla modalità scelta e alle richieste dell'utente.

4.3.2 Core Application

Come visto in precedenza, la GUI si interfaccia col **Core** richiedendo l'operazione di esecuzione di un particolare test. Tale operazione non è elementare ma può essere suddivisa in 5 fasi :

- Preparazione ambiente
- Configurazione ambiente
- Avvio demoni
- Esecuzione test
- Uccisione demoni e chiusura terminali

Quindi il Core deve espletare un certo numero di funzionalità, come mostrato dal seguente diagramma dei casi d'uso:

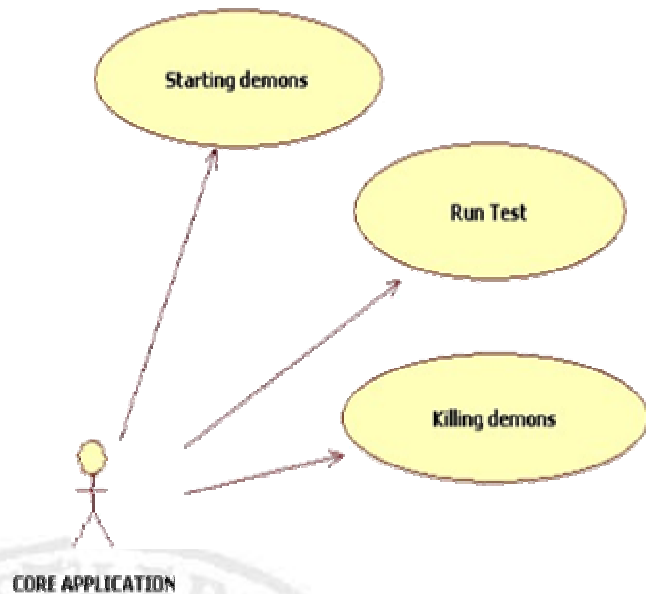


Figura 4.8 Use Case Diagram del Core

Per espletare tali funzionalità, il Core eseguirà una lunga serie di operazioni. L'utente non dovrà quindi più svolgere tali operazioni manualmente ma dovrà soltanto restare in attesa che l'esecuzione dello specifico test termini.

Scendendo ad un maggior livello di dettaglio, possiamo pensare il Core come suddiviso in un certo numero di componenti che interagiscono tra loro per espletare le funzionalità elencate nel precedente diagramma :

- **CORE MAIN**, che coordina e gestisce le fasi di preparazione e avvio dei test;
- **TERMINAL**, che rappresenta un singolo terminale di lavoro su cui si eseguono le istruzioni necessarie all'avvio dei test;

- **SERVICE**, che offre i servizi di basso livello ai terminali;
- **TEST**, che rappresenta lo specifico test da mandare in esecuzione.

Il diagramma risultante è il seguente :

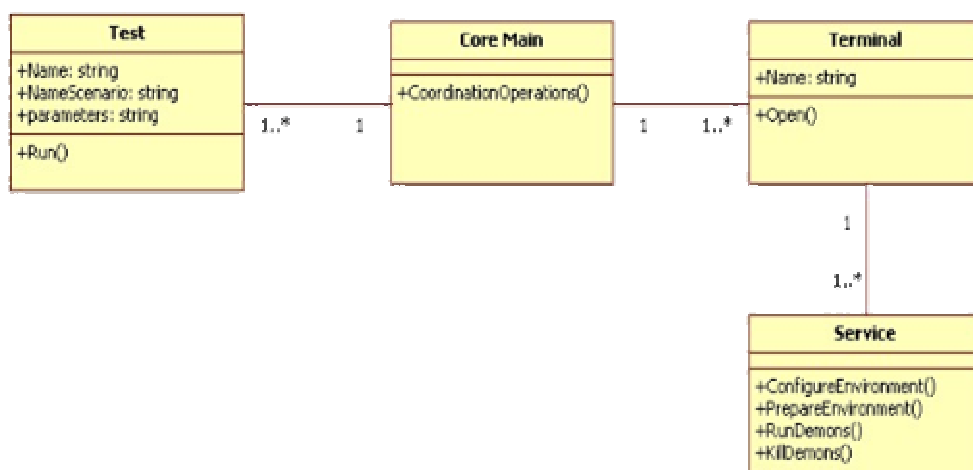


Figura 4.9 Class Diagram delCore

Per comprendere in che modo le varie classi comunicheranno tra loro per espletare le funzionalità richieste è possibile utilizzare il seguente Sequence Diagram :

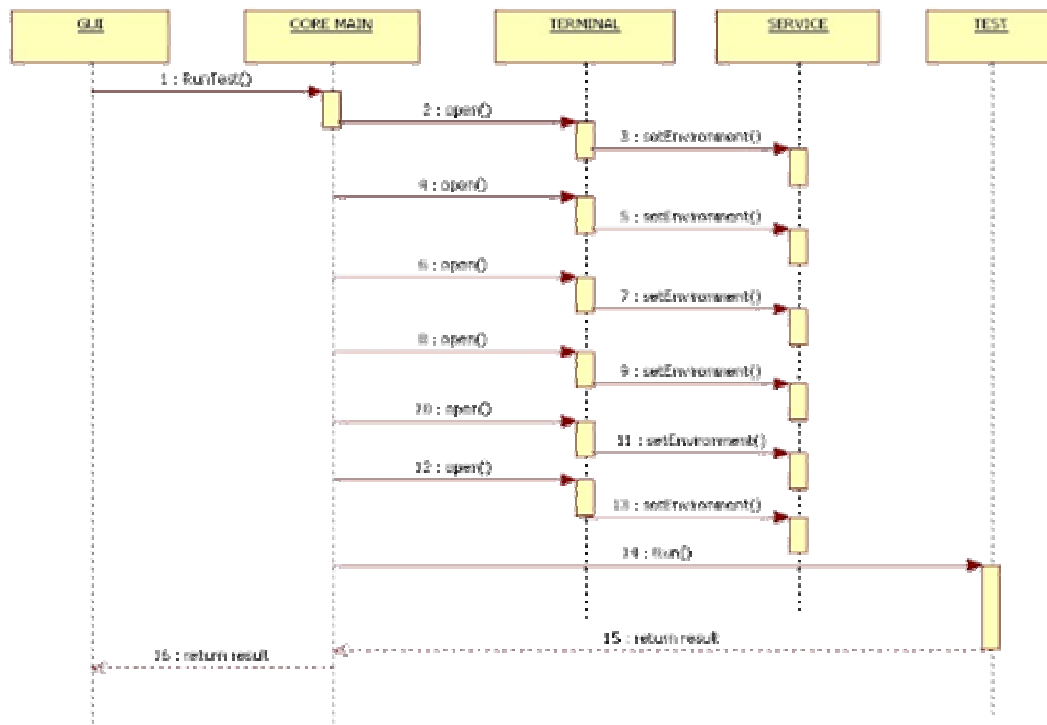


Figura 4.10 Sequence Diagram del Core

4.3.3 Class Diagram

Una volta suddivisi i due macrocomponenti in componenti elementari, il diagramma delle classi risultante è il seguente :

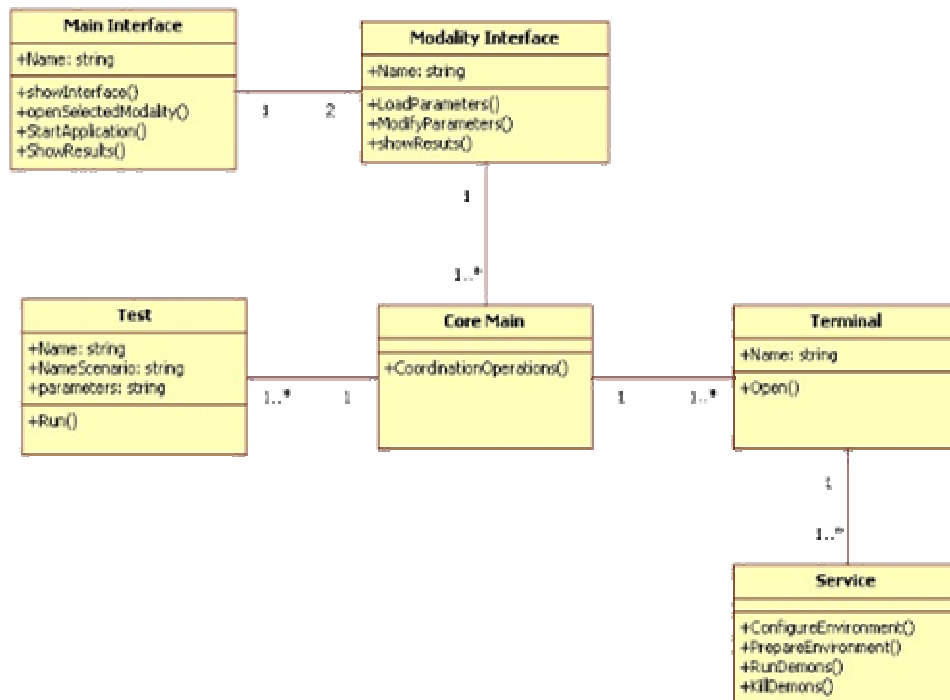


Figura 4.11 Class Diagram definitivo

Dal diagramma precedente è possibile individuare le seguenti classi :

- **Main Interface**, che rappresenta l'interfaccia di avviamento del tool da sviluppare;
- **Modality Interface**, che indica l'interfaccia relativa alla modalità di avviamento scelta;
- **Core Main**, che coordina e gestisce le fasi di preparazione e avvio dei test;
- **Test**, che fa riferimento ad un singolo test di qualifica;
- **Terminal**, che rappresenta un terminale di lavoro su cui si eseguono tutte le operazioni di preparazione ambiente e avvio del test;

- **Service**, che offre servizi di basso livello al terminale.

Nel diagramma avremmo potuto specificare anche la classe **Scenario**, che rappresenta lo scenario di esecuzione di uno specifico test. Essa, però, è legata alla classe Test da una relazione uno a uno, quindi possiamo accorpate le due classi inserendo l'attributo "*NomeScenario*" all'interno della classe Test.

Le classi appena elencate sono collegate tra loro dalle seguenti associazioni :

- **MainInterface e ModalityInterface**, con molteplicità uno a due, poiché un'interfaccia presenta due modalità di funzionamento le quali ovviamente fanno riferimento alla stessa interfaccia;
- **ModalityInterface e CoreApplication**, con molteplicità uno a molti, necessaria per l'interfacciamento tra GUI e Core;
- **CoreMain e Terminal**, con molteplicità uno a molti, in quanto il core si occupa della gestione e del coordinamento delle operazioni da eseguire su più terminali;
- **CoreMain e Test**, con molteplicità uno a molti, necessaria per la gestione dell'esecuzione del test;
- **Terminal e Service**, con molteplicità uno a molti, che associa ad ogni singolo terminale uno o più servizi di base necessari per il corretto svolgimento delle operazioni di preparazione ed esecuzione del test.

Capitolo 5

Implementazione del tool “QUALTTOOL”

Il seguente capitolo ha lo scopo di descrivere le scelte ed i dettagli implementativi del tool “QUALTTOOL” (Qualification Testing Tool)

5.1 Individuazione di un pattern procedurale comune

Dall'analisi dei casi d'uso presentati nel capitolo precedente, si può facilmente osservare che ognuno di essi richiede un'esecuzione automatica dei test di qualifica. Per garantire tale automatizzazione, si è osservato che il 50% dei test ha in comune tutte le operazioni necessarie per le fasi di preparazione e avvio. Un primo passo, quindi, è quello di individuare un pattern procedurale comune a tali test in modo da automatizzarne le fasi di avvio e preparazione riducendo notevolmente sia il lavoro dell'utente che il tempo necessario all'effettiva esecuzione.

Tale pattern può essere riassunto dal seguente schema:

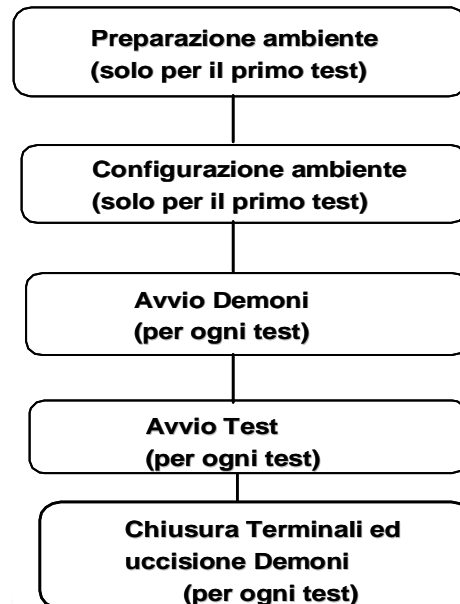


Figura 5.1 Pattern

Esso consente quindi di eseguire tali operazioni in maniera del tutto automatica, attraverso l'avvio di opportuni script di shell che saranno analizzati in dettaglio successivamente.

Presentiamo ora l'organizzazione del tool e gli aspetti implementativi.

5.2 Organizzazione di “QUALTTOOL”

Il tool per la verifica software di un sistema distribuito, a cui è stato dato il nome di “QUALTTOOL” (Qualification Testing Tool), è costituito da un package “QualificationTester” contente:

- un'interfaccia Java, responsabile dell'interazione con l'operatore;
- file di script di shell necessari al corretto funzionamento del tool; tali file vengono richiamati automaticamente dall'interfaccia Java a seconda delle operazioni che l'utente desidera eseguire;
- una cartella "*QualificationTesterRemoto*", che viene copiata sulla macchina remota durante le fasi di preparazione e avvio del test; tale cartella contiene degli script che devono essere eseguiti sulla macchina remota e che sono fondamentali per un corretto funzionamento del tool;
- un file di configurazione "*config.xml*", necessario per il corretto caricamento dei parametri di input di default all'avvio di ambedue le modalità di funzionamento dell'interfaccia, la cui struttura sarà analizzata nei paragrafi successivi;
- un file di testo "*test.txt*" contenente l'elenco dei test automatizzabili;
- un file di testo "*scenari.result*" all'interno del quale verranno memorizzati i risultati dei test eseguiti.

La scelta dello scripting di shell è nata dalla forte presenza di comandi linux nella fase di preparazione e lancio di un test, mentre la scelta del linguaggio di programmazione Java è dovuta ad una lunga serie di vantaggi riportati di seguito:

- ***semplice***: può essere programmato senza la necessità di lunghi corsi di addestramento;
- ***object-oriented***: il linguaggio è progettato per essere orientato agli oggetti da cima a fondo;
- ***familiare***: la sintassi del linguaggio JAVA è molto simile a quella del C++ e questo lo rende un linguaggio familiare;

- ***robusto e sicuro***: il linguaggio è progettato per creare software altamente affidabile;
- ***indipendente dalla piattaforma e portabile***: il linguaggio è progettato per supportare applicazioni che verranno distribuite sulle più eterogenee piattaforme senza la necessità di dover riscrivere il programma per i vari ambienti.

Analizziamo ora in dettaglio le scelte implementative per i vari componenti del tool.

5.2.1 QUALTTOOL Interface

L'utilizzo di un'interfaccia Java per la realizzazione del tool nasce con lo scopo di semplificare il lavoro dell'utente, che, come visto in precedenza, è costretto ad effettuare una lunga serie di operazioni prima di poter effettivamente lanciare un test di qualifica. Grazie alla realizzazione di tale interfaccia, l'utente viene sollevato dall'onere di eseguire tali istruzioni, ma esso avrà a disposizione dei semplici comandi per il lancio dei test. Tutta la fase di avvio dei test sarà quindi eseguita dal Core dell'applicazione con le modalità che vedremo in seguito.

Dall'analisi dei requisiti effettuata in precedenza, si è visto che al tool è richiesta una doppia modalità di funzionamento:

- ***manuale***, per l'esecuzione di un singolo test di qualifica;
- ***automatica***, per l'esecuzione di un'intera lista di test di qualifica.

Per soddisfare i requisiti appena elencati, si è scelto di dare all'interfaccia la seguente struttura:

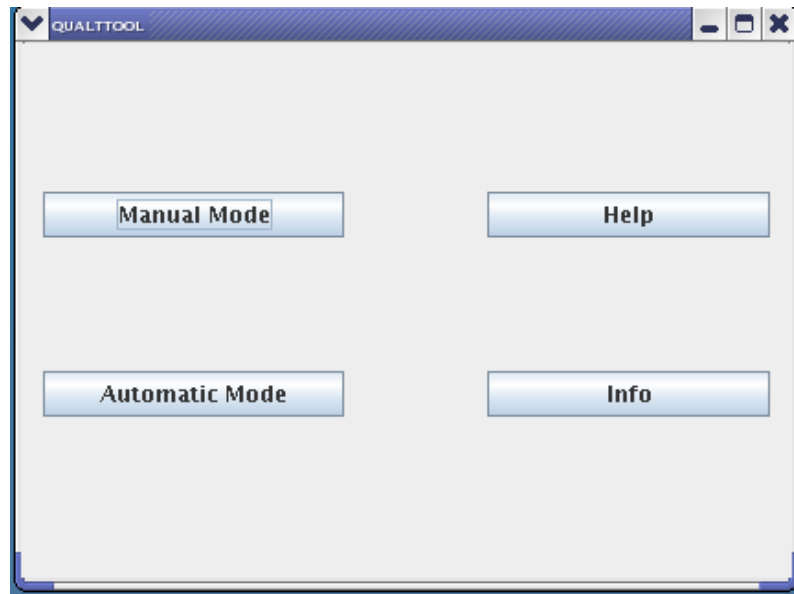


Figura 5.2 Interfaccia QUALTTOOL

L'interfaccia di avviamento del tool presenta quindi i seguenti comandi :

- **ManualMode**, che rappresenta la modalità di avviamento manuale del tool;
- **AutomaticMode**, che permette di avviare la modalità automatica del tool;
- **Help e Info**, che forniscono rispettivamente un aiuto al funzionamento del tool e delle informazioni di carattere generale sul tool sviluppato.

Analizziamo ora in dettaglio le scelte implementative per ciascuna delle due modalità di avviamento del tool.

5.2.1.1 Manual Mode

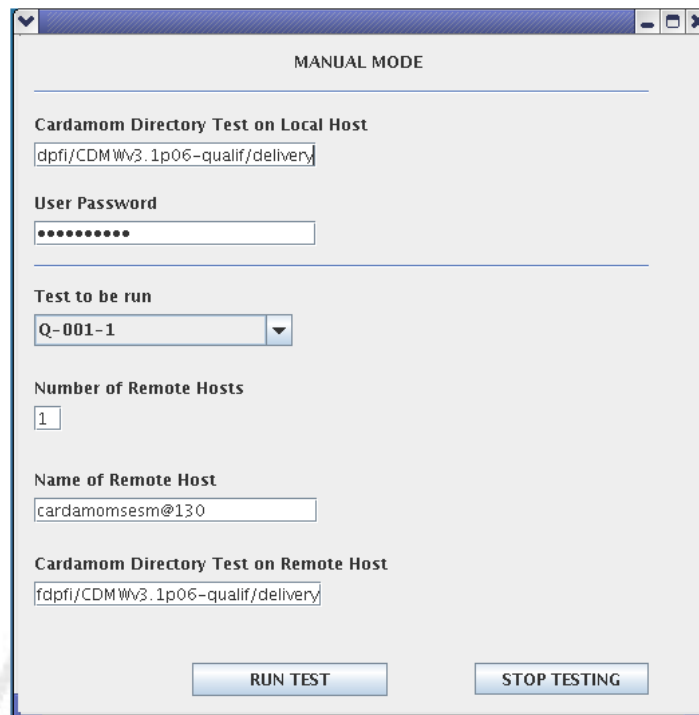
La modalità manuale nasce con lo scopo di soddisfare i seguenti requisiti :

- automatizzare le fasi di preparazione e lancio di un singolo test di qualifica;
- eliminare l'interazione utente-macchina durante l'esecuzione di uno specifico test;
- corretto posizionamento dei terminali di lavoro per assicurare all'utente una

chiara comprensione delle operazioni svolte dalla macchina durante l'esecuzione di uno specifico test;

- riempimento dei menù dei test tramite file di configurazione.

All' avvio, la modalità manuale si presenta nel seguente modo :



The screenshot shows a window titled "MANUAL MODE". Inside, there are several sections:

- Cardamom Directory Test on Local Host**: A text field containing "dpfi/CDMWv3.1p06-qualif/delivery".
- User Password**: A text field with masked characters (dots).
- Test to be run**: A dropdown menu currently showing "Q-001-1".
- Number of Remote Hosts**: A text input field containing the number "1".
- Name of Remote Host**: A text field containing "cardamomseem@130".
- Cardamom Directory Test on Remote Host**: A text field containing "fdpfi/CDMWv3.1p06-qualif/delivery".

At the bottom of the window, there are two buttons: "RUN TEST" and "STOP TESTING".

Figura 5.3 Manual Mode

Analizziamola in dettaglio:

- **Cardamom Directory Test on Local Host**, rappresenta la directory di installazione dei test di qualifica della piattaforma sulla macchina locale;
- **User Password**, indica la password di utente sulla macchina locale;
- **Test to be run**, rappresenta il particolare test di qualifica da mandare in esecuzione; l'utente può scegliere tale test utilizzando un apposito menù a tendina;

- **Number of Remote Hosts**, che permette di selezionare il numero di hosts coinvolti nell'attività di testing; il tool realizzato consente solo l'esecuzione di test che prevedono una configurazione con due host, tuttavia ci sono casi di test che richiedono uno scenario di esecuzione con più di due host;
- **Name of Remote Host**, che indica il nome dell'host remoto coinvolto nell'attività di testing;
- **Cardamom Directory Test on Remote Host**, rappresenta la directory di installazione dei test di qualifica della piattaforma sulla macchina remota;
- **RunTest**, che invia al Core la richiesta di esecuzione del test specificato.

Per consentire all'utente di avere uno strumento completamente automatico, tali parametri vengono prelevati da un file di configurazione "**config.xml**" la cui struttura è la seguente :

```
<root>
  <localdir>/home/fdpfi/CDMWv3.1p06-qualif/delivery</localdir>
  <remotedir>/home/fdpfi/CDMWv3.1p06-qualif/delivery</remotedir>
  <testfilename>/home/fdpfi/CDMWv3.1p06-qualif/delivery/share/QualificationTester/test.txt</testfilename>
  - <hosts maxHostNumber="5" defaultHostNumber="1">
    <hostname>cardamomsesm@130</hostname>
    <hostname>testbed4</hostname>
  </hosts>
</root>
```

Figura 5.4 File "**config.xml**"

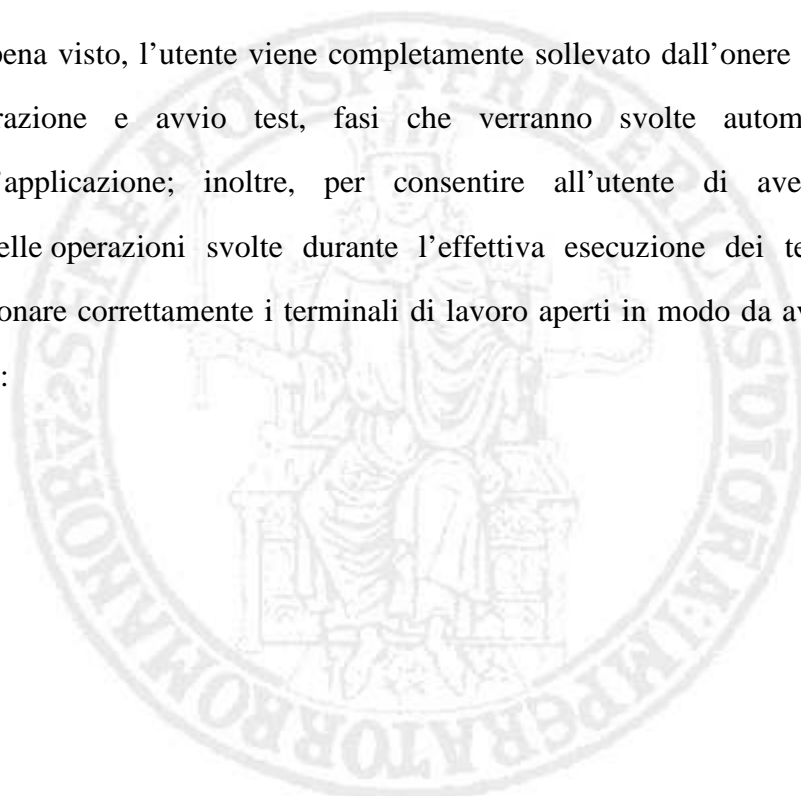
In questo modo, all'avvio della modalità manuale, i parametri necessari al lancio del test vengono caricati automaticamente; l'utente, quindi, dovrà soltanto selezionare lo

specifico test da avviare e premere sul pulsante “**RunTest**”; tuttavia esso può anche modificare i parametri di default direttamente all’interno dell’interfaccia.

Una volta immessi correttamente i parametri (eventuali immissioni errate o mancanti dei parametri saranno segnalate dal tool con opportuni messaggi di errore) e premuto su “**RunTest**”, l’interfaccia richiede al Core dell’applicazione di eseguire tutte le operazioni necessarie alla preparazione e avvio del test; verranno quindi passati i parametri necessari al Core il quale eseguirà una serie di script di shell secondo le modalità che vedremo nel paragrafo successivo.

Inoltre, l’interfaccia mette a disposizione dell’utente un comando “**StopRunning**” che viene utilizzato nel caso in cui si desidera terminare il test durante la sua esecuzione, comando che può essere utilizzato anche per terminare definitivamente un test la cui esecuzione non è andata a buon fine.

Come appena visto, l’utente viene completamente sollevato dall’onere di eseguire le fasi di preparazione e avvio test, fasi che verranno svolte automaticamente dal Core dell’applicazione; inoltre, per consentire all’utente di avere una chiara visione delle operazioni svolte durante l’effettiva esecuzione dei test, si è scelto di posizionare correttamente i terminali di lavoro aperti in modo da avere la seguente situazione:



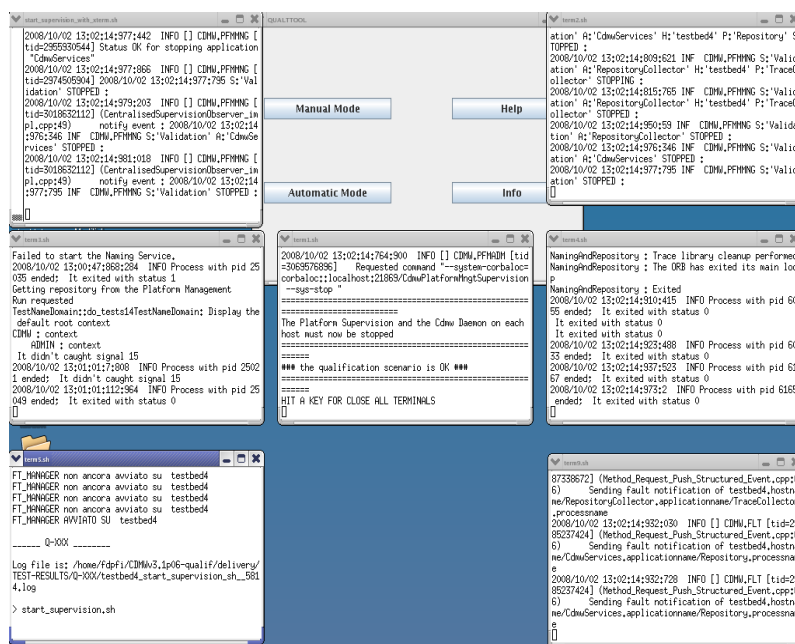


Figura 5.5 Posizionamento automatico dei terminali di lavoro

Grazie alle scelte implementative appena descritte, si è riusciti a soddisfare tutti i requisiti evidenziati in fase di progettazione; questo per quanto riguarda la modalità di funzionamento manuale del tool.

Analizziamo ora la modalità di avviamento automatica e le scelte implementative ad essa associate.

5.2.1.2 Automatic Mode

La modalità automatica nasce con lo scopo di soddisfare i seguenti requisiti :

- automatizzare le fasi di preparazione e lancio di un'intera lista di test di qualifica;
- eliminare l'interazione utente-macchina durante l'esecuzione dei test;
- presenza di una repository dei risultati;
- riempimento dei menù dei test tramite file di configurazione.

All'avvio, la modalità automatica si presenta all'utente nel seguente modo:

The screenshot shows a window titled "AUTOMATIC MODE". Inside, there are several input fields and buttons:

- Cardamom Directory Test on Local Host:** A text box containing the path `e/fdpfi/CDMWv3.1p06-qualif/delivery`.
- User Password:** A text box with masked characters (dots).
- Number of Remote Hosts:** A text box containing the number `1`.
- Name of Remote Host:** A text box containing `cardamomsesm@130`.
- Cardamom Directory Test on Remote Host:** A text box containing the path `/home/fdpfi/CDMWv3.1p06-qualif/delivery`.
- Select file that contains tests to be run:** A section with a **Select File** button and a text box containing the file path `CDMWv3.1p06-qualif/delivery/share/QualificationTester/test.txt`.
- RUN TESTS:** A button located below the file selection section.
- A large empty rectangular box at the bottom of the window.

Figura 5.6 Automatic Mode

I parametri sono uguali a quelli analizzati precedentemente per la modalità manuale, con l'unica differenza che, poiché tale modalità viene utilizzata per eseguire un'intera lista di test, l'utente dovrà selezionare, anziché lo specifico test, un file di testo contenente l'elenco dei test da mandare in esecuzione. Anche per tale modalità di funzionamento del tool i parametri di default vengono prelevati dal file di configurazione "*config.xml*".

L'utente, dopo aver selezionato il file da cui prelevare i test da eseguire, deve

soltanto cliccare su “RunTests”; sarà poi l’interfaccia (dopo aver controllato la correttezza dei parametri inseriti) a comunicare col Core, in maniera trasparente all’utente, per richiedere l’esecuzione dei test. Le modalità e gli script richiamati dal Core per eseguire tali operazioni saranno analizzati nel paragrafo successivo.

Un requisito fondamentale richiesto alla modalità automatica è la presenza di una repository dei risultati al termine dell’esecuzione della lista di test specificata. Per garantire il soddisfacimento di tale requisito, la scelta implementativa fatta è stata quella di visualizzare, al termine dell’esecuzione dei test, una schermata contenente, per ogni riga, il particolare test eseguito e l’esito dell’esecuzione (“OK” test andato a buon fine, “NOK” test fallito). Per rendere più comprensibile quanto detto, mostriamo la schermata che appare all’utente al termine del lancio dei test :

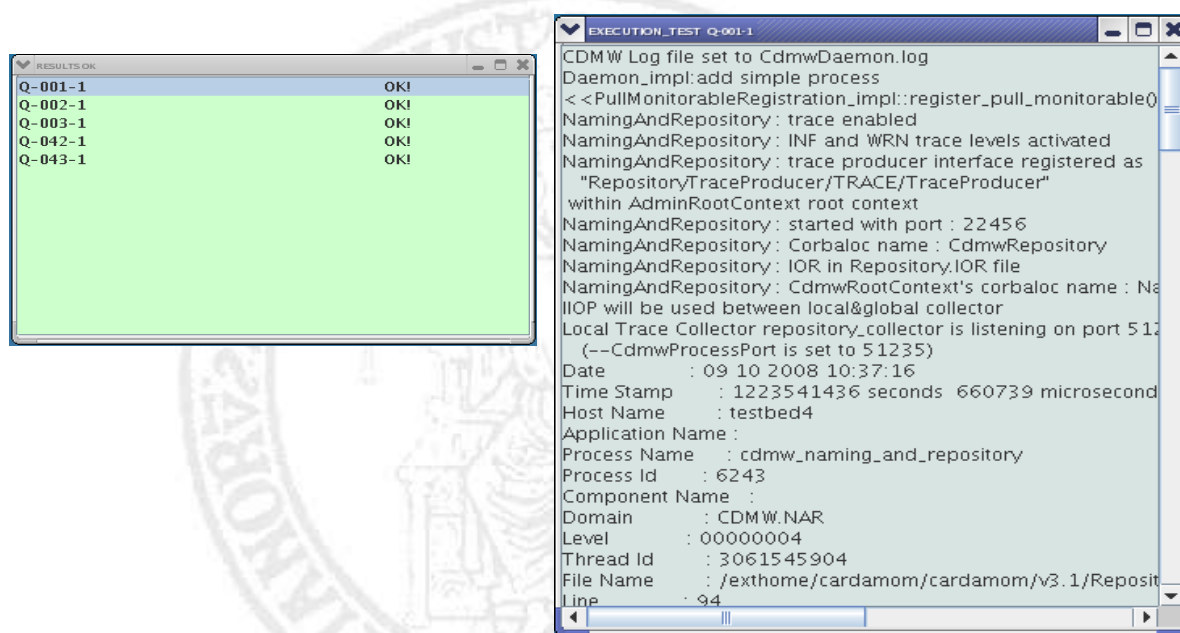


Figura 5.7 Repository dei risultati

La schermata di destra di figura 5.7 compare nel caso in cui l'utente selezioni col mouse uno specifico test per visualizzarne i dettagli di esecuzione.

Grazie alle scelte implementative appena descritte, si è riusciti a soddisfare tutti i requisiti evidenziati in fase di progettazione per quanto riguarda la modalità di avviamento automatica del tool.

5.2.2 Core Application

Abbiamo precedentemente detto che sarà l'interfaccia a comunicare col Core per richiedere l'esecuzione dei test di qualifica. Tale interazione avviene mediante l'avvio di appositi file di script di shell.

Per l'implementazione del Core dell'applicazione si è scelto di utilizzare il linguaggio di shell bash in quanto il nostro applicativo verrà utilizzato in ambiente Linux.

Per semplificare la descrizione dei vari script conviene suddividerli in quattro categorie:

- script richiamati in fase di avviamento di QUALTTOOL e responsabili della gestione delle operazioni da eseguire e dell'avvio dei terminali di lavoro;
- script responsabili delle azioni dei terminali di lavoro ossia della preparazione dell'ambiente e dell'avvio dei vari demoni;
- script che forniscono le funzioni di base agli script precedenti;
- script che hanno il compito di uccidere i demoni avviati e chiudere i terminali aperti al termine dell'esecuzione di ogni singolo test avviato.

Analizziamo in dettaglio ciascuna delle categorie appena elencate.

5.2.2.1 Avviamento di QUALTTOOL

Alla richiesta di esecuzione test da parte dell'interfaccia, il Core dell'applicazione avvierà uno dei seguenti script in base alla modalità di avviamento del tool:

- “*startautom.sh*” in caso di modalità di avviamento automatica;
- “*startmanual.sh*” se si utilizza la modalità di funzionamento manuale.

Entrambi ricevono gli input dall'interfaccia ed hanno in comune le seguenti operazioni :

- copia della cartella “*QualificationTesterRemoto*” sulla macchina remota utilizzata per l'attività di testing, cartella che contiene gli script necessari per le operazioni da eseguire in remoto;
- avvio di sei terminali di lavoro mediante la funzione “*terminale*”:

terminale “*term2.sh*”;

terminale “*term3.sh \$HOST_NAME_1 \$HOST_DIR_1 \$PASS*”;

terminale “*term4.sh \$PASS*”

terminale “*term5.sh*”;

terminale “*term9.sh*”;

terminale “*term1.sh \$TEST \$SCRIPT_ESECUZIONE \$PASS \$CDMW_DIR*”;

La funzione “*terminale*” ha un comportamento diverso per le due modalità di funzionamento:

- per la **modalità automatica**, i vari terminali di lavoro non vengono mostrati all'utente; ciò si ottiene utilizzando il comando bash *"iconic"* che permette di ridurre i terminali ad icona e non mostrarli quindi all'utente;
- per la **modalità manuale**, i vari terminali, come visto precedentemente, vengono aperti in maniera da consentire all'utente di avere una chiara visione delle operazioni svolte dalla macchina per l'avvio ed esecuzione dei test; per disporre in maniera ordinata i terminali di lavoro, si è utilizzata l'opzione *"geometry"* a seguito della quale occorre specificare la dimensione e la posizione dello specifico terminale.

5.2.2.2 Preparazione ambiente e avvio demoni

Tale categoria è composta dai seguenti file di script di shell:

"term1.sh", *"term2.sh"*, *"term3.sh"*, *"term4.sh"*, *"term5.sh"*,
"term9.sh". Vediamo ora come si comportano tali file:

- ***"term1.sh"***: prepara l'ambiente richiamando appositi script, verifica se il platform observer (necessario per un corretto funzionamento dei test di *qualifica*) sia avviato sulla macchina e lancia il test. Al termine dell'esecuzione del test, dopo aver presentato i risultati all'utente, avvia uno script responsabile della chiusura dei terminali di lavoro e dell'uccisione demoni.
- ***"term2.sh"***: prepara l'ambiente, verifica che il *"platform supervision"* (necessario per i test di *qualifica*) sia avviato sulla macchina e lancia il *"platform observer"*.

- **“term3.sh”**: avvia la fase di preparazione al test sulla macchina remota richiamando gli script “term3_2.sh” e “term3_3.sh” precedentemente copiati in remoto.
- **“term4.sh”**: prepara l’ambiente con gli accessi di root.
- **“term5.sh”**: prepara l’ambiente, verifica che il servizio di “fault tolerance” (necessario per i test di *qualifica*) sia avviato sulla macchina ed avvia il “platform supervision”.
- **“term9.sh”**: prepara l’ambiente e verifica se il “platform daemon” (necessario per i test di *qualifica*) sia avviato sulla macchina.

La preparazione dell’ambiente avviene richiamando appositi script che ora andremo ad analizzare.

5.2.2.3 Servizi base per la preparazione dell’ambiente

Come visto, ogni terminale di lavoro, esegue come prima azione una preparazione dell’ambiente di lavoro; tale operazione è ottenuta tramite l’avvio di opportuni script:

“servizio_locale.sh” e **“servizio_locale_root.sh”**.

Il file **“servizio_locale.sh”** prepara l’ambiente per la macchina locale su cui si esegue il tool, mentre il file **“servizio_locale_root.sh”** compie le stesse operazioni del file **“servizio_locale.sh”**, ma con gli accessi di root.

5.2.2.4 Chiusura terminali e uccisione demoni

Abbiamo precedentemente detto che il `term1`, una volta completata l'esecuzione del test, avvia uno script che si occupa di chiudere i terminali di lavoro aperti e uccidere i demoni avviati. Tale script, *"startkill.sh"*, è costituito dalle seguenti righe di codice:

- `echo $PASSW | sudo -S pkill cdmw`
- `echo $PASSW | sudo -S pkill xterm > xterm`

dove *"PASSW"* è la password di utente sulla macchina locale.

5.3 Sincronizzazione Demoni

Così come evidenziato in fase di analisi, vi è la necessità di avviare i demoni secondo la seguente sequenza:

1. Avvio del platform daemon
2. Avvio del fault manager
3. Avvio del supervision
4. Avvio dell'observer
5. Avvio del test

Una prima soluzione introdotta al fine di soddisfare questo requisito prevedeva l'utilizzo del comando `sleep`, che introduceva un ritardo, ritenuto sufficiente per l'avvio di ciascun daemon.

Questa soluzione si è rivelata errata, in quanto a causa delle imprevedibili condizioni di carico delle macchine su cui vengono avviati i demoni, il ritardo introdotto non garantiva l'avvio dei demoni nella sequenza corretta.

A tal punto è stata studiata una diversa soluzione che ritarda l'avvio di un daemon fin tanto che il demone che lo precede nella sequenza sia sicuramente avviato. Questa informazione viene acquisita dai file di log prodotti dalla piattaforma cardamon.

In questo modo, si è ottenuta una soluzione indipendente dalle condizioni di carico delle macchine su cui viene lanciato il tool.

5.4 Obiettivi raggiunti

Dopo aver presentato tutte le scelte implementative per soddisfare i requisiti individuati in fase di analisi, vediamo quali sono stati gli obiettivi raggiunti dal tool sviluppato:

- Possibilità da parte dell'utente di eseguire un intero elenco di test (*AutomaticMode*) oppure un singolo test (*ManualMode*) senza interagire con la macchina.
- Riduzione del tempo di esecuzione dei test: se si considera un singolo test si ha un risparmio di tempo di circa 12 minuti (primo test) o 5 minuti (test successivi); dunque, si passa da un totale di circa 6 ore ad un totale di circa 2 ore, con un risparmio di tempo complessivo pari a **4 ORE**.
- Presenza di una repository dei risultati (*AutomaticMode*) e chiara visione delle operazioni svolte durante l'esecuzione di uno specifico test (*ManualMode*).
Presentazione dei risultati in una forma più leggibile.

Capitolo 6

Risultati Sperimentali

In questo capitolo vengono illustrati i risultati maturati durante il processo di studio e realizzazione del tool.

6.1 TEST AUTOMATIZZABILI

Per quanto riguarda la definizione dell'insieme di test automatizzabili è stata realizzato un'esame accurato dei test.

Tra i test presi in considerazione (v3.1), una parte risultava essere costituito da Ispezioni e Dimostrazioni, così come riportato in tabella :

NOME TEST	Tipologia
Q-005	ISPEZIONE
Q-008	ISPEZIONE
Q-009	ISPEZIONE
Q-010	ISPEZIONE
Q-011	ISPEZIONE
Q-013	ISPEZIONE
Q-015	ISPEZIONE
Q-019	ISPEZIONE
Q-025	ISPEZIONE
Q-032	ISPEZIONE
Q-036	ISPEZIONE

Q-046	DIMOSTRAZIONE
Q-051	ISPEZIONE
Q-052	ISPEZIONE
Q-056	ISPEZIONE
Q-063	ISPEZIONE
Q-078	ISPEZIONE
Q-080	ISPEZIONE
Q-081	ISPEZIONE
Q-083	DIMOSTRAZIONE
Q-084	DIMOSTRAZIONE
Q-099	ISPEZIONE
Q-109	DIMOSTRAZIONE
Q-111	ISPEZIONE
Q-201	ISPEZIONE
Q-202	ISPEZIONE
Q-203	ISPEZIONE
Q-204	ISPEZIONE
Q-205	ISPEZIONE
Q-206	ISPEZIONE
Q-207	ISPEZIONE
Q-208	ISPEZIONE
Q-209	ISPEZIONE
Q-210	ISPEZIONE
Q-211	ISPEZIONE
Q-212	ISPEZIONE
Q-213	ISPEZIONE
Q-214	ISPEZIONE
Q-215	ISPEZIONE
Q-216	ISPEZIONE
Q-217	ISPEZIONE

Questi per la propria natura, non risultano automatizzabili in quanto richiedono di andare ad ispezionare codice oppure il contenuto di specifici file di piattaforma.

Tra i test rimanenti, solo in parte risultavano conformi al pattern procedurale standard.

In tabella è riportato l'insieme di test automatizzabili.

NOME TEST
Q-001
Q-002
Q-003
Q-004
Q-017
Q-018
Q-042
Q-043
Q-044
Q-045
Q-053
Q-055
Q-058
Q-064
Q-089
Q-095
Q-096
Q-100
Q-105
Q-110
Q-112
Q-113
Q-115
Q-116
Q-117
Q-118
Q-119
Q-120
Q-122
Q-124
Q-129
Q-131
Q-133
Q-134
Q-135
Q-342

A partire da questa lista è stato costruito il file **test.txt** utilizzato dal tool. Ovviamente in questo vengono inseriti i test comprensivi dei relativi scenari.

Successivamente alla realizzazione del tool, si è cercato di capire per quali motivi i rimanenti test non risultassero automatizzabili.

In tabella riportiamo quei test che risultano aderenti alla procedura standard eccezion fatta per l'ultimo passo, in cui prevedono l'uso di un ulteriore parametro o la modifica dell'ultima istruzione.

NOME TEST	Parametro aggiuntionale
Q-047	exectime
Q-061	exectime
Q-082	exectime
Q-087	exectime
Q-101	exectime
Q-102	exectime
Q-141	exectime
Q-300	Keystop
Q-301	Keystop
Q-302	Keystop
Q-303	Keystop
Q-304	Keystop
Q-310	ccm
Q-311	ccm
Q-312	ccm
Q-313	ccm
Q-330	Keystop
Q-332	Keystop
Q-333	Keystop
Q-334	Keystop
Q-336	Keystop

In tabella, troviamo quei test non conformi alla procedura standard, che a differenza dei precedenti stravolgono la procedura magari richiedendo interazione con il tester durante la vera e propria esecuzione del test.

NOME TEST
Q-006
Q-007
Q-022
Q-023
Q-024
Q-027
Q-028
Q-029
Q-030
Q-031
Q-033
Q-034
Q-035
Q-037
Q-038
Q-039
Q-040

Q-045
Q-048
Q-049
Q-050
Q-054
Q-057
Q-058
Q-059
Q-060
Q-062
Q-065
Q-066
Q-067
Q-068
Q-069
Q-070
Q-071
Q-072
Q-073
Q-074
Q-075
Q-076
Q-077
Q-079
Q-085
Q-086
Q-088
Q-090
Q-091
Q-092
Q-093
Q-094
Q-097
Q-098
Q-104
Q-106
Q-121
Q-123
Q-125
Q-126
Q-127
Q-128
Q-130
Q-132
Q-136
Q-137
Q-138
Q-139
Q-340
Q-400
Q-401
Q-402
Q-403
Q-404
Q-405

6.2 Tempi di Testing

Lo realizzazione del tool ha richiesto un'analisi dei test di qualifica attualmente previsti dalla piattaforma. Durante tale processo è stato necessario comprendere la procedura manuale, evidenziando il notevole tempo richiesto per il lancio di un test. In particolare :

Fase della procedura di lancio	Tempo (in minuti)	
Preparazione ambiente di test	7	
Avvio demoni	1	
Esecuzione Test	3	
Uccisione demoni	1	
	12	TOT

Questo ci ha consentito di valutare la tempistica relativa all'esecuzione dell'intero test set. In conclusione :

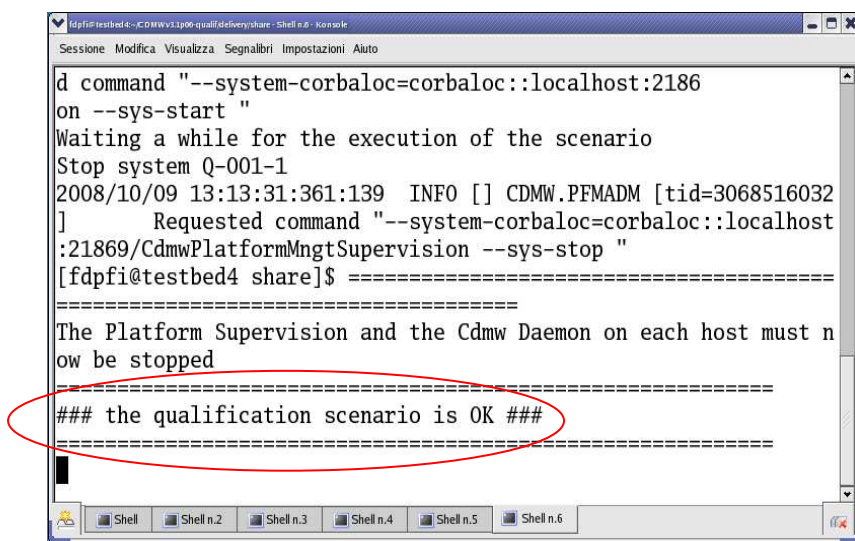
- un tempo di circa 12 minuti per il primo test ;
- un tempo di circa 5 minuti per ognuno dei successivi.

Con QUALTTTOOL si è ottenuta una riduzione dei tempi di testing di circa il **70%**.

Infatti, se per eseguire il set di test (automatizzabile) con procedura manuale erano necessari circa 320 minuti, con il tool è stato sufficiente un tempo di circa 90 minuti.

6.3 Osservazioni sulla procedura di valutazione

Una volta realizzato il tool per l'automatizzazione dei test di qualifica, è stato richiesto di stabilire se la procedura di valutazione fosse significativa o meno. Per meglio comprendere il problema, diciamo che il tool deriva l'esito del test (visualizzato nel report RESULT_OK) dalle informazioni di log prodotte dalla piattaforma (evidenziate in figura).



```
fdpfi@testbed4:~/CDMWv1.1p06qualifdelvnyshare$ Shell n.6 - karole
Sessione Modifica Visualizza Segnalibri Impostazioni Aiuto

d command "--system-corbaloc=corbaloc::localhost:2186
on --sys-start "
Waiting a while for the execution of the scenario
Stop system Q-001-1
2008/10/09 13:13:31:361:139 INFO [] CDMW.PFMADM [tid=3068516032
] Requested command "--system-corbaloc=corbaloc::localhost
:21869/CdmwPlatformMngtSupervision --sys-stop "
[fdpfi@testbed4 share]$ =====
=====
The Platform Supervision and the Cdmw Daemon on each host must n
ow be stopped
=====
### the qualification scenario is OK ###
=====
```

Figura 6.1: Esito di un Test

Durante alcune delle prove effettuate, a seguito del rilevamento messaggi sospetti riportati negli output dei demoni lanciati, siamo andati ad analizzare in maniera dettagliata i log prodotti dalla piattaforma.

Si è riscontrato, che alcuni test ritenuti eseguiti con successo (in quanto si aveva il msg “ the qualification scenario is ok”), in realtà non erano stati eseguiti oppure erano stati eseguiti in parte. Questo riscontro era possibile solo andando ad ispezionare i log prodotti dalla piattaforma. In definitiva la procedura di valutazione del test è da ritenersi *inaffidabile*.

Da qui si è ritenuto necessario comprendere le cause di questa inaffidabilità e quindi si è proceduto cercando di capire quale script di piattaforma aveva la responsabilità di eseguire la valutazione.

Da un'analisi degli script di shell, si è giunti alla conclusione che tale script è il

RUN_QUALIFICATION_TEST.sh.

Prima di descrivere tale valutazione, dobbiamo dire che la piattaforma produce molti file di log tra cui:

- **CdmwQualificationTests.output** : output del platform daemon.

Lo script, una volta eseguito il test , provvede a costruire un'ulteriore log in cui vengono accorpati i **CdmwQualificationTests.output** relativi ad entrambi gli host coinvolti nel test.

Detto log è il seguente:

- **GlobalCdmwQualifictionTests.output** : in cui vengono riportati gli output dei terminali term3 (platform daemon HOST1) e term4 (platform daemon HOST2).

La valutazione consiste nel verificare la presenza di eventuali FAILED o FAILURE nel log GLOBALE (GlobalCdmwQualifictionTests.output). Queste informazioni di log, potrebbero non essere prodotte o perché non previste per lo specifico test oppure a causa di un fallimento.

In conclusione, il TOOL automatizza la fase di Testing, almeno per quanto riguarda preparazione ed avvio dei test, ma è comunque necessario che l'operatore ispezioni i log per comprendere l'effettivo esito del Test (vedi figura).

Nel paragrafo successivo descriveremo l'ispezione manuale evidenziandone problematiche e limiti.

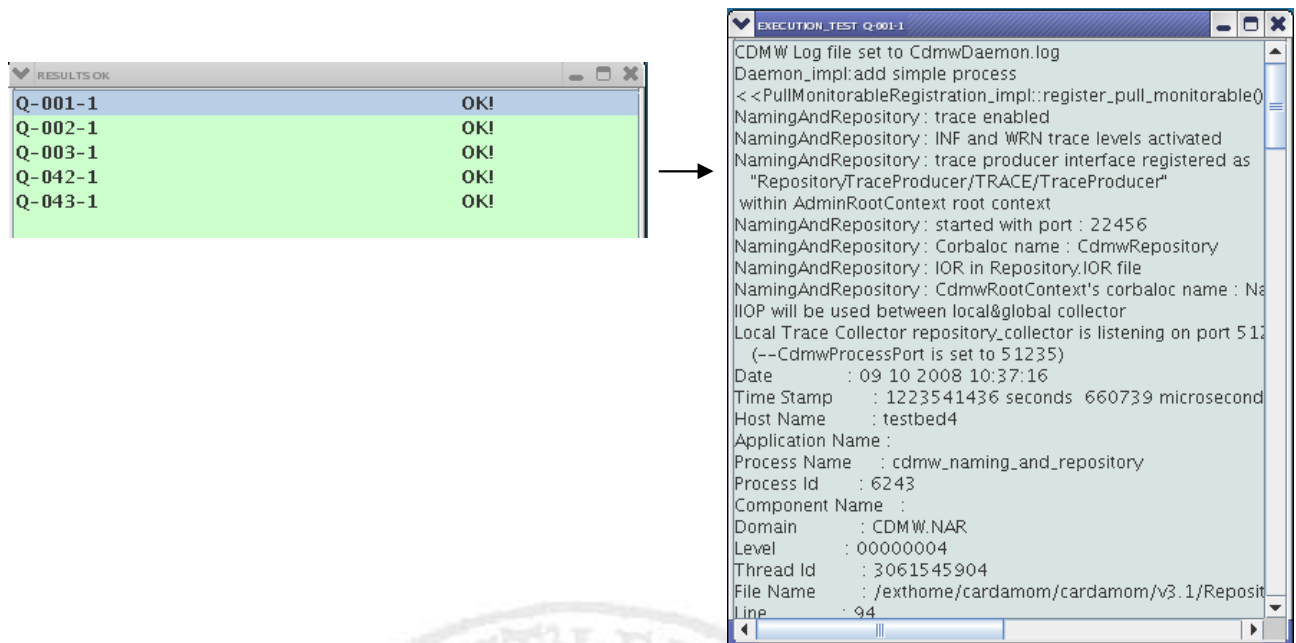
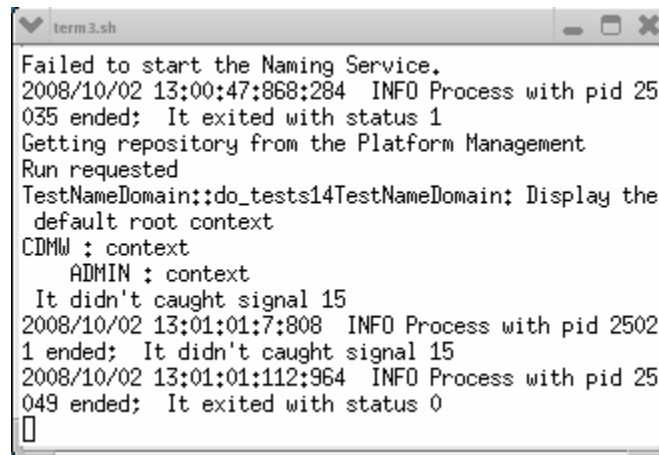


Figura 6.2 Valutazione dei log

6.3.1 Ispezione manuale

Così come accennato in precedenza, la valutazione di un test richiede che l'operatore ispezioni i file di log relativi ai platform daemon lanciati sui due host. Questo non sempre mi garantisce di comprendere l'effettivo esito di un test.

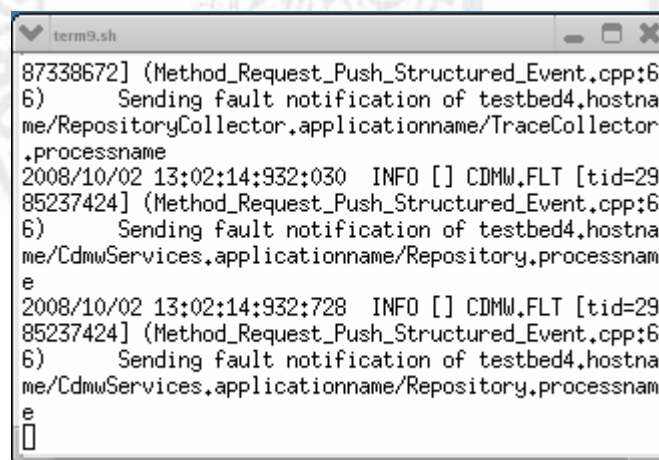


```
term3.sh
Failed to start the Naming Service.
2008/10/02 13:00:47:868:284 INFO Process with pid 25
035 ended; It exited with status 1
Getting repository from the Platform Management
Run requested
TestNameDomain::do_tests14TestNameDomain: Display the
default root context
CDMW : context
ADMIN : context
It didn't caught signal 15
2008/10/02 13:01:01:7:808 INFO Process with pid 2502
1 ended; It didn't caught signal 15
2008/10/02 13:01:01:112:964 INFO Process with pid 25
049 ended; It exited with status 0
```

Figura 6.3 Log platform daemon (term3)

Per rendere la valutazione più affidabile dovremmo ispezionare anche i log relativi agl'altri demoni coinvolti nella procedura di testing

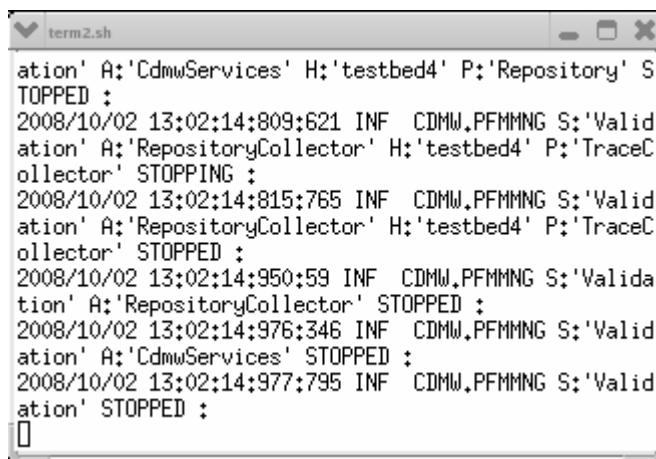
Ad esempio il log del demone FT_Manager, in cui vengono riportate molte informazioni tra cui quelle relative ad eventuali fault rilevati.



```
term9.sh
87338672] (Method_Request_Push_Structured_Event.cpp:6
6) Sending fault notification of testbed4.hostna
me/RepositoryCollector.applicationname/TraceCollector
.processname
2008/10/02 13:02:14:932:030 INFO [] CDMW.FLT [tid=29
85237424] (Method_Request_Push_Structured_Event.cpp:6
6) Sending fault notification of testbed4.hostna
me/CdmwServices.applicationname/Repository.processnam
e
2008/10/02 13:02:14:932:728 INFO [] CDMW.FLT [tid=29
85237424] (Method_Request_Push_Structured_Event.cpp:6
6) Sending fault notification of testbed4.hostna
me/CdmwServices.applicationname/Repository.processnam
e
```

Figura 6.4 Log FT_Manager daemon

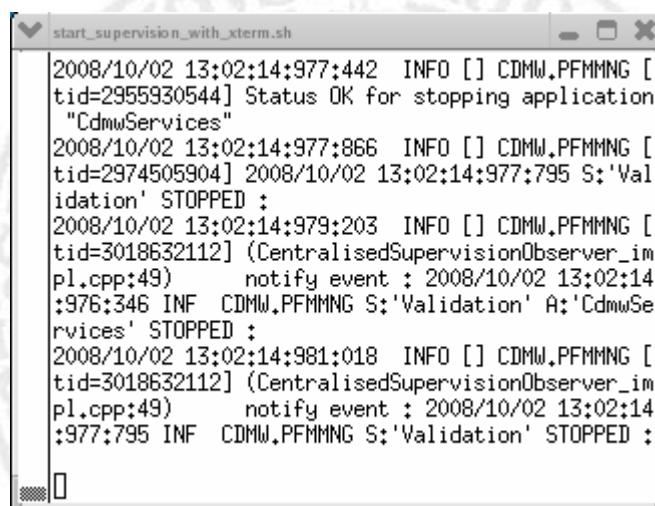
Ulteriori informazioni possono essere reperite nel log dell'Observer che contiene eventuali allarmi e cambiamenti di stato dei processi, delle applicazioni e dei nodi.



```
ation' A:'CdmwServices' H:'testbed4' P:'Repository' S  
TOPPED :  
2008/10/02 13:02:14:809:621 INF CDMW.PFMMNG S:'Valid  
ation' A:'RepositoryCollector' H:'testbed4' P:'TraceC  
ollector' STOPPING :  
2008/10/02 13:02:14:815:765 INF CDMW.PFMMNG S:'Valid  
ation' A:'RepositoryCollector' H:'testbed4' P:'TraceC  
ollector' STOPPED :  
2008/10/02 13:02:14:950:59 INF CDMW.PFMMNG S:'Valida  
tion' A:'RepositoryCollector' STOPPED :  
2008/10/02 13:02:14:976:346 INF CDMW.PFMMNG S:'Valid  
ation' A:'CdmwServices' STOPPED :  
2008/10/02 13:02:14:977:795 INF CDMW.PFMMNG S:'Valid  
ation' STOPPED :
```

Figura 6.5 Log Observer daemon

Altra fonte di informazione potrebbe essere il log del Supervision.



```
2008/10/02 13:02:14:977:442 INFO [] CDMW.PFMMNG [  
tid=2955930544] Status OK for stopping application  
"CdmwServices"  
2008/10/02 13:02:14:977:866 INFO [] CDMW.PFMMNG [  
tid=2974505904] 2008/10/02 13:02:14:977:795 S:'Val  
idation' STOPPED :  
2008/10/02 13:02:14:979:203 INFO [] CDMW.PFMMNG [  
tid=3018632112] (CentralisedSupervisionObserver_im  
pl.cpp:49) notify event : 2008/10/02 13:02:14  
:976:346 INF CDMW.PFMMNG S:'Validation' A:'CdmwSe  
rvices' STOPPED :  
2008/10/02 13:02:14:981:018 INFO [] CDMW.PFMMNG [  
tid=3018632112] (CentralisedSupervisionObserver_im  
pl.cpp:49) notify event : 2008/10/02 13:02:14  
:977:795 INF CDMW.PFMMNG S:'Validation' STOPPED :
```

Figura 6.6 Log Supervision daemon

Da quanto detto si evidenziano complessità e notevole durata del processo di valutazione di un test e quindi si rende necessaria una soluzione automatica, di cui parleremo nel successivo pragrafo.

6.3.2 Proposte di Soluzione

Potremmo rendere tutti i test di qualifica conformi ad uno specifico criterio di log. Per fare questo è necessario ristrutturare l'intera parte di testing prevista per la piattaforma middleware utilizzata.

Una soluzione di questo tipo richiederebbe di far in modo che la piattaforma produca un summary (resoconto) del test realizzato su ciascun host (così come viene fatto per una parte dei test) e quindi derivare l'esito del test dai report dei summary.

Un'altra possibile soluzione potrebbe consistere in:

- Produrre una repository dei log “SIGNIFICATIVI”, cioè log che possiamo ritenere corrispondenti ad esecuzioni sicuramente corrette (derivati da un'analisi manuale dei log).
- Modificare la procedura di valutazione, in modo che l'esito del test sia ottenuto mediante confronto tra il log prodotto e quello ad esso corrispondente nella repository.

Data la notevole quantità di informazioni riportate sui log, realizzare questo confronto con approccio tradizionale richiederebbe dei tempi abbastanza onerosi. Un approccio alternativo è quello di utilizzare un classificatore realizzato con metodologie di intelligenza artificiale.

Questa soluzione è realizzabile solo se la sequenza dei messaggi nei log è sempre la stessa, cioè l'ordine dei messaggi rimane invariato da esecuzione ad esecuzione.

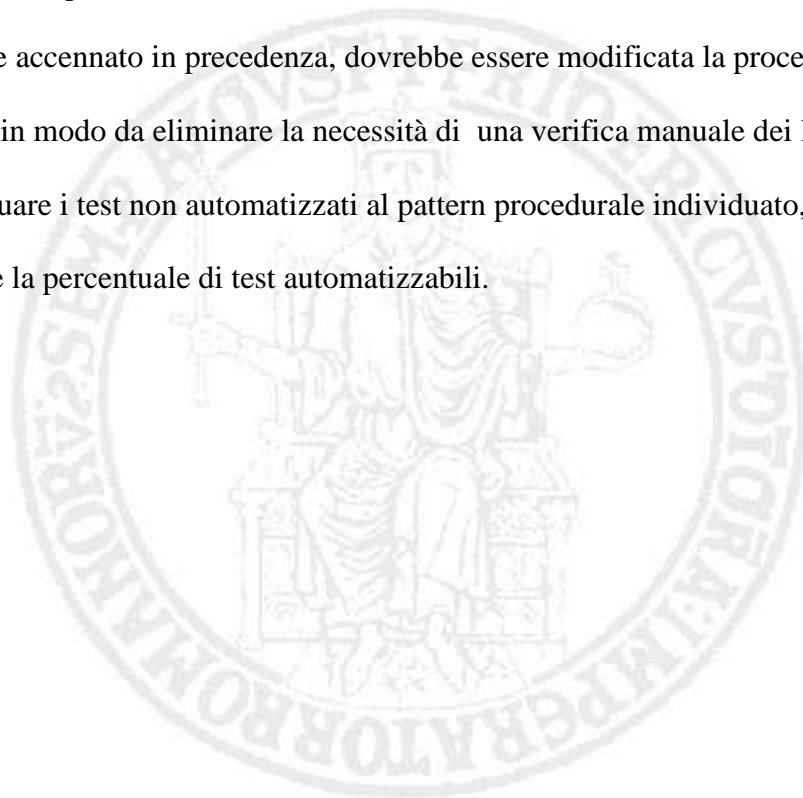
Conclusioni e Sviluppi futuri

Il tool realizzato consente una riduzione dei tempi di testing di circa il 70% in riferimento ai test automatizzabili pari al 50% dell'intero test set. Consente di eseguire un singolo test con una procedura molto più semplice e veloce.

Nonostante tali vantaggi, a causa dell'inaffidabilità della procedura di valutazione, è necessario che l'operatore umano (responsabile del testing) analizzi i log dei test in modo da comprenderne l'effettivo esito.

Il tool dovrebbe essere esteso in modo da automatizzare anche i test che richiedono un numero di host superiori a 2.

Inoltre, come accennato in precedenza, dovrebbe essere modificata la procedura di valutazione, in modo da eliminare la necessità di una verifica manuale dei log dei test ed inoltre, adeguare i test non automatizzati al pattern procedurale individuato, in modo da incrementare la percentuale di test automatizzabili.



Bibliografia

Robyn R. Lutz “Software Engineering for Safety: A Roadmap”

<http://www.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finallutz.pdf>

A.Avizienis, J.Laprie, B. Randell , C. Landwehr “Basic Concepts and Taxonomy of Dependable and Secure Computing”

C. Knutson and S. Carmichael 2000 “Safety first: avoiding software mishaps”

W.R.Dunn 2003 “Designing Safety Critical Computer Systems”

Wen-Der Jiang 1997 “ Experience Of Applying Corba Middleware To Air Traffic Control Automation Systems”

S.Russo, C.Savy, D.Cotroneo, A.Sergio 2002 “Introduzione aCORBA”

Leszek A. Maciaszek 2002 “Sviluppo di sistemi informativi con UML”

Martin Fowler 2004 “UML Distilled”

Roger S. Pressman 2004 “Principi di Ingegneria del software”

Thales, Selex-SI “Cardamom, Product overview”:

http://cardamom.objectweb.org/docs/CARDAMOM_PROV_Rev03.doc

Thales, Selex-SI “Technical Services Tutorial for Cardamom”:

http://cardamom.objectweb.org/docs/CARDAMOM_TS_Tutorial_Rev04.doc

Thales, Selex-SI “Software Test Plan / Description / Report for SYSTEM MANAGEMENT”:

http://cardamom.objectweb.org/docs_SMG_STPDR_Rev05

Thales, Selex-SI “Qualification Test Plan / Description / Report”:

http://cardamom.objectweb.org/docs/CARDAMOM_QTPDR_Rev_07

