Università degli Studi dell'Insubria

FACOLTA' DI SCIENZE MATEMATICHE
FISICHE E NATURALI

Corso di Laurea Specialistica in Informatica

Definizione e sviluppo di un tool per la generazione automatica di documentazione di testing nel contesto di progetti software open source

Tesi di Laurea di:
JACOPO EMOROSO

Matricola:
610428

Relatore:
Prof. SANDRO MORASCA
Prof. LUIGI LAVAZZA
Correlatore:
Dott. DAVIDE TOSI
Dott. DAVIDE TAIBI

Anno Accademico 2008-2009

Riassunto

Il presente lavoro di tesi si colloca in un filone di attività che corrispondono ad interessi comuni del progetto QualiPSo (Quality Platform for Open Source Software), un progetto integrato che ha l'intenzione di contribuire allo stato dell'arte del Software Open Source (OSS) definendo e implementando tecnologie, procedure e policies per influenzare le attuali pratiche di sviluppo OSS.
In particolare, si è posta l'attenzione sullo stato dell'arte dell'attività di testing del software, considerando le diverse fasi che costituiscono il processo, i documenti di test che caratterizzano ognuna di esse, ed i problemi legati alla loro generazione, specialmente in ambienti OSS.

Lo scopo di questa tesi è la creazione di un modello semplice e intuitivo, in grado di assistere gli utenti durante l'intero processo di testing, fornendo un supporto concreto per la generazione e l'aggiornamento della documentazione di test. Oltre a fronteggiare i problemi legati al testing di OSS mediante l'automatizzazione del processo, favorendo l'aggregazione, la standardizzazione e la condivisione dei dati, si vogliono elaborare soluzioni specifiche per le diverse tipologie di test, come quelli di unità, di integrazione e di regressione.

Prima della stesura del nuovo modello è stata effettuata un'attenta analisi degli strumenti degli ambienti di lavoro esistenti, che supportano la generazione automatica di documentazione di test, mettendo in evidenza i limiti in un ambiente come quello dell'OSS, dove il codice e le metodologie per il testing appaiono in modo frammentario e poco uniforme. Inoltre, da questa analisi è emersa l'importanza di mantenere il forte legame che unisce e mette in relazione la documentazione generata

nelle varie fasi del processo di testing, dalla progettazione, all'implementazione, all'esecuzione e valutazione dei test.

Le problematiche evidenziate hanno spinto alla creazione del modello T-Doc, in grado di snellire e facilitare la generazione e l'aggiornamento della documentazione di testing, sfruttando in modo automatico la struttura del codice sorgente e le informazioni in esso contenute. T-Doc è in grado di agire in tutte le fasi del processo di testing, aggregando tutte le informazioni relative alla progettazione, implementazione, risultati di esecuzione dei casi di test, e suggerimenti specifici per test di integrazione e regressione.

T-Doc è composto da tre layer: 1) Documentazione Automatica dei Casi di Test, 2) Suggerimenti Automatici per Test di Integrazione e Regressione, 3) Documentazione Automatica dei Report di Esecuzione dei Test.

Dopo aver definito il modello, è stata progettato il tool T-Doc definendo l'architettura ed il funzionamento di tutti e tre i layer ed implementando il primo.

Si sono voluti verificare da subito i vantaggi introdotti dalla documentazione automatica dei casi di test, applicando gli standard definiti per il modello T-Doc a MacXim, un progetto OSS sviluppato presso il Dipartimento di Informatica e Comunicazione dell'Università dell'Insubria. Tale verifica ha permesso di affermare la validità del modello T-Doc, comprendendo più a fondo i reali benefici introdotti e le debolezze del tool. I restanti due layer sono in fase di implementazione e anch'essi saranno validati nell'ambito del progetto Macxim.

Nel prossimo futuro si vogliono estendere ed ampliare i casi reali su cui validare l'efficacia del modello T-Doc, in modo che esso possa affermarsi come standard per il testing dell'OSS.

INDICE

1.

CAPITOLO PRIMO

INTRODUZIONE

1.1 CONTESTO APPLICATIVO

L'attività di testing del software nasce con la comparsa della prima riga di codice.

E' un concetto che si sviluppa negli anni, evolve nelle metodologie, introduce cambiamenti nelle tecniche di sviluppo e nell'idea stessa di software.

L'analisi dell'attività di testing ha evidenziato una stretta relazione tra ogni fase del processo e la relativa documentazione generata al suo interno. Tali documenti si sono dimostrati di importanza fondamentale per una buona riuscita del testing di un prodotto software.

Nonostante questo, sono numerosi i problemi che ostacolano la creazione della documentazione di test e la mancanza di essa determina un rallentamento nella diffusione di tali prodotti.

Ciò è ancora più vero nell'ambito di Software Open Source, dove lo sviluppo avviene grazie alla collaborazione di più persone e spesso esistono difficoltà di coordinamento, frammentazione del codice, mancanza di omogeneità e standardizzazione. In questo dominio, la documentazione gioca un ruolo fondamentale, essendo al tempo stesso la causa e la soluzione a queste problematiche.

Riconsiderando tali affermazioni, sulla base delle conoscenze teoriche, metodologiche e delle diverse tipologie di testing del software, si vuole introdurre un nuovo modello per affrontare il problema della documentazione dell'attività di testing.

Tale modello vuole essere una soluzione concreta all'ingente uso di risorse richieste da questa attività, aiutando gli sviluppatori ad eseguire con scrupolo ed attenzione ogni singola fase del processo di documentazione, in grado di avere dei risvolti qualitativamente rilevanti sul prodotto software finale.

Per studiare l'applicabilità ed i reali benefici di questa soluzione e perché essa possa diventare uno standard di uso comune nel testing di Software Open Source, si vuole applicare tale modello ad un progetto reale.

1.2 OBIETTIVI DELLA TESI

Questa tesi ha come oggetto la creazione di un tool che semplifichi la generazione e l'aggiornamento della documentazione tecnica in relazione all'attività di testing del software.

In particolare, si vuole fornire un supporto completo alla documentazione dell'intero processo di testing, sia per l'estrazione dei report dei risultati di test, che per i dati di progettazione ed implementazione, elaborando soluzioni specifiche per diverse tipologie di test, come quelli di unità, di integrazione e di regressione.

Tali dati vogliono essere aggregati e condivisi attraverso un repository centrale dedicato.

La soluzione proposta vuole fronteggiare efficacemente i problemi che si verificano durante la generazione della documentazione di test.

Il tool realizzato deve essere in grado di affrontare tali problematiche, specialmente quando esse si verificano con maggior forza durante il testing di Software Open Source.

Infine, si vuole validare la soluzione proposta attraverso l'implementazione del tool in un progetto Open Source reale, per osservarne il comportamento durante la sua esecuzione, comprenderne le reali potenzialità e trarre delle conclusioni sui vantaggi introdotti dall'adozione di tale modello.

1.3 STRUTTURA DELLA TESI

Questa tesi è strutturata in sei capitoli.

Il primo capitolo consiste in un'introduzione al lavoro svolto, dove viene brevemente presentato il contesto applicativo nel quale ci si è trovati ad operare, vengono definiti gli obiettivi, si definisce l'approccio con il quale è stato affrontato il problema e si traccia la struttura della tesi.

Il secondo capitolo riguarda il testing del software, ed introduce i concetti che stanno alla base di questa attività.
Si ripercorre brevemente la storia e l'evoluzione del concetto di testing del software, partendo dagli albori fino ad arrivare agli anni '90, in cui è stato definito e riadattato il puzzle teorico e modellistico che è oggi alla base del testing di un applicativo.
Dopo aver delineato gli obbiettivi e l'importanza di quest'attività si sono discusse le principali tecniche di testing: l'analisi statica, che risulta essere associata alla forma, alla struttura e alla documentazione di un'applicazione; l'analisi dinamica, invece basata sulla sua esecuzione.
Vengono poi presentate le principali tipologie di testing del software: il testing di unità, di integrazione, di sistema, di accettazione e di regressione.

Nel terzo capitolo viene illustrato il processo di testing e la documentazione prodotta all'interno di ognuna delle fasi che lo caratterizzano.
Si è poi evidenziato lo stretto legame che intercorre tra i documenti di test e l'importanza da loro ricoperta in funzione di una buona riuscita del processo di testing.

Inoltre, sono stati presi in esame i motivi che spingono a sottovalutare il ruolo della documentazione di testing e le problematiche legate alla produzione di tali documenti durante lo sviluppo di progetti software Open Source.

Il quarto capitolo riguarda lo sviluppo del tool per la generazione automatica della documentazione di test e i vantaggi offerti da questa soluzione, sia per snellirne e facilitarne la creazione e l'aggiornamento, che come supporto all'intero processo di testing.
Sono stati esaminati tool e framework già esistenti, che supportano la generazione automatica di documenti, analizzandone pregi e limiti nel contesto di testing.
In particolare è stato studiato il tool Javadoc, che nonostante non sia idoneo e adattabile all'intero processo di testing, presenta interessanti caratteristiche di estensibilità.
E' stato poi introdotto un nuovo tool studiato appositamente per rispondere alle esigenze della generazione di documentazione di test, chiamato T-Doc.
Sono stati evidenziati i motivi che hanno portato alla sua realizzazione, i vantaggi introdotti con questo nuovo modello, la sua architettura, il suo funzionamento e la struttura dei layer che lo compongono.

Nel capitolo cinque si è voluto adottare l'approccio T-Doc ad un progetto Open Source reale, al fine di validare il modello precedentemente descritto, comprenderne più a fondo il funzionamento e ottenere riscontri effettivi sui benefici e vantaggi introdotti.
Si è quindi descritto il processo di testing che ha portato all'implementazione della test suite per tale progetto, chiamato Macxim,

che è stata pianificata e progettata applicando gli standard definiti per il modello T-Doc.
Sono quindi stati messi in evidenza i risultati che hanno sancito la validità e la bontà dell'approccio proposto.

Infine, nel capitolo sei, sono state tratte le conclusioni del lavoro svolto e si è riflettuto su quanto emerso, evidenziando quali potrebbero essere gli sviluppi futuri nell'applicazione di tale modello a progetti reali.

2.

CAPITOLO SECONDO

TESTING DEL SOFTWARE

Il testing è un processo fondamentale nel ciclo di vita del software, che è nato e si è sviluppato con la prima riga di codice scritta nella storia.
In questo capitolo, in primo luogo si definisce il concetto di testing e si illustrano le sue caratteristiche principali.
Viene definito il contesto storico in cui il concetto di testing nasce, ripercorrendo la sua evoluzione dai primi anni '50 e la crescente introduzione di nuove metodologie che hanno contribuito a quella che è oggi la definizione di testing.
Successivamente si analizzerà quali sono i principali obiettivi di quest'attività e perché oggi il testing ricopre un'importanza fondamentale nella produzione del software.

Si discuteranno poi i due principali approcci che possono essere adottati per il testing del software: l'analisi statica, che risulta essere associata alla forma, alla struttura e alla documentazione di un'applicazione; l'analisi dinamica, invece basata sulla sua esecuzione.
Verranno quindi prese in esame anche le principali tipologie di testing del software, come: test di unità, test di integrazione, test di sistema, test di regressione, test di accettazione, test di performance, stress test, test di sicurezza e quelle fasi che precedono il rilascio di un prodotto software comunemente chiamate alfa e beta testing.

2.1 CONCETTI BASE

Il testing del software è un procedimento utilizzato per individuare le carenze di correttezza, completezza e affidabilità di un prodotto software in corso di sviluppo [1].
Con tale attività si vuole quindi assicurare la qualità del prodotto tramite la ricerca di difetti, ovvero una sequenza di istruzioni e procedure che, quando eseguiti con particolari dati di input e in particolari ambienti operativi, generano dei malfunzionamenti.
Un malfunzionamento è un comportamento del software non atteso da parte dell'utente, quindi difforme dalle specifiche e dai requisiti impliciti o espliciti definiti per tale applicazione.
Lo scopo del testing è quindi quello di rilevare i difetti tramite i malfunzionamenti, in modo da ridurre al minimo la probabilità che tali malfunzionamenti si possano verificare nel normale utilizzo del prodotto software.

Il testing non può stabilire che un prodotto funzioni correttamente sotto tutte le possibili condizioni di esecuzione, ma può evidenziare difetti sotto specifiche condizioni.

Infatti, data l'impossibilità di testare tutte le combinazioni di input e i possibili ambienti software e hardware in cui l'applicazione può trovarsi ad operare, la probabilità di malfunzionamenti non può essere ridotta a zero, ma deve essere ridotta al minimo, in modo da risultare accettabile per l'utente.

L'accettabilità di una certa probabilità di malfunzionamento dipende dal tipo di applicazione che si vuole realizzare.

In questo contesto pare evidente che in un applicazione "life-critical", ad esempio in ambito ospedaliero o militare, dove un malfunzionamento può mettere a rischio la vita umana, la qualità richiesta sarà molto più elevata di quella attesa per software di ufficio o un videogioco [a,b].

Il testing viene definito come "processo di pianificazione, preparazione, esecuzione e analisi, con lo scopo di stabilire le caratteristiche di un sistema informativo, e di dimostrare le differenze tra lo stato attuale e lo stato richiesto" [2].

Tale definizione mette in luce che l'attività di testing non può essere limitata alla ricerca di difetti nel codice sorgente tramite il manifestarsi di malfunzionamenti, ma deve essere estesa ad un esame più approfondito dei requisiti definiti per l'applicazione e la loro implementazione.

Alcuni difetti, infatti, si manifestano sottoforma di mancanza di requisiti, inadeguatezza degli stessi o comunque sono generati in modo indipendente dalla correttezza o meno del codice sorgente.

2.2 STORIA ED EVOLUZIONE DEL TESTING

2.2.1 – Gli Albori

Il concetto di testing, parallelamente all'evoluzione del software, è mutato nel tempo, definendo progressivamente nuovi obiettivi e generando nuove tecniche.

L'attività di testing nasce e trova le sue radici nella prima riga di codice software scritta nella storia.

Sin da subito si è reso necessario un controllo di correttezza di quanto scritto nel codice applicativo, dando vita ad un attività di testing embrionale orientata soprattutto al debugging, la correzione degli errori.

La definizione data al testing in questo periodo è ben espressa da questa frase: "il testing è ciò che un programmatore fa per individuare e correggere i bugs nei propri programmi" [1].

La separazione concettuale tra debugging e testing di un applicativo appaiono quindi come un'operazione unitaria, in cui il solo fine da perseguire nella fase di test è quello della ricerca dell'errore per poterlo correggere [3].

Nel 1950 Alan Turing scrive il famoso articolo "Computing machinery and intelligence" sulla rivista Mind, che da molti è considerato il primo articolo sul testing di programmi.

Infatti, al suo interno viene posta la domanda "Come possiamo sapere se un programma si dimostra intelligente?", che in altri termini, se l'intento è quello di scrivere tale programma, la domanda potrebbe essere un caso particolare di "Come possiamo sapere se un programma soddisfa i suoi requisiti?".

Per affrontare la questione Turing propone un test in cui il comportamento del programma in oggetto deve essere indistinguibile dal

comportamento di un essere umano, agli occhi di una persona esterna che non sia a conoscenza della reale natura dei soggetti interrogati.
Tale test in realtà simula il comportamento di un programmatore durante l'attività di testing, che confronta i risultati dell'esecuzione vera e propria del programma con quelli attesi.

2.2.2 – Anni '50, '60 e '70

Tra la fine degli anni '50 e la fine degli anni '70 l'idea del testing muta profondamente, portando ad una nuova visione.
Grazie a Charles Baker il testing viene chiamato nel 1957 con il nome di "program checkout", ovvero il controllo del programma, identificando in tale operazione due principali obbiettivi: assicurarsi che il programma funzioni e assicurarsi che il programma risolva il problema.
I due scopi vengono spesso tradotti e riassunti con l'assicurarsi che siano soddisfatti i requisiti e successivamente si tenta di fornire dei fondamenti teorici ed approcci sistematici all'attività di testing, che si separa da quella di debugging.
Tuttavia in questo periodo è posto un forte accento sul voler dimostrare la correttezza dei programmi attraverso i test.
Negli anni successivi emerge che questa visione porta con sé diversi limiti e addirittura non può neanche essere messa in pratica.
In quello che è chiamato "Demonstration-Oriented Period" [3], il periodo orientato alla dimostrazione, un test perfetto si ha quando il programma non contiene errori.
Gli anni '70 vedono il diffondersi dell'idea che un software possa essere testato in modo esaustivo, quindi che possa sopravvivere la dimostrazione della correttezza di un programma grazie ai test.

Proprio per tale ragione hanno luogo diversi studi e ricerche sul path coverage testing, che prevede l'esame di tutte le linee di codice e tutte le possibili sequenze di esecuzione di un programma.
Ancora oggi tale tecnica è ritenuta la più completa ed esaustiva, tuttavia dato l'elevato costo in termini di tempo e l'inapplicabilità a certi tipi di problemi, viene usata raramente e limitatamente a piccole porzioni di codice.
Esemplare per comprendere l'epoca il lavoro di Goodhough e Gerhart del 1975, in cui si parla di "testing esaustivo definito sia in termini di sequenze di esecuzione del programma che di dominio di input del programma".
Proprio in questo periodo ci sono notevoli conquiste non solo sul piano teorico, ma anche una ricca produzione di metodi e tecniche di testing, la loro classificazione e toponomastica, lo sviluppo di modelli per la rappresentazione dei programmi e la classificazione degli errori.
Tuttavia il sogno di un testing esaustivo, sufficiente a dimostrare la correttezza di un programma, viene presto infranto.

2.2.3 – Anni '80

Il libro scritto da Myers nel 1979, dal titolo "The Art of Software Testing", fornisce le basi per tecniche di testing più efficienti e rivoluziona completamente il concetto di test, segnando di fatto l'inizio di una nuova epoca.
Per la prima volta il testing del software è descritto come "il processo di esecuzione di un programma con l'intento di trovare errori".
Proprio da questo concetto il nome di "Destruction-Oriented Period", ovvero il periodo orientato alla distruzione, in cui i casi di test assumono un valore maggiore in base al numero di errori che riescono a trovare.
Mentre in precedenza inconsciamente potevano essere scelti dati di test con bassa probabilità di causare errori, in questo periodo c'è un

ribaltamento della situazione, marcato dalla voglia di voler trovare e mostrare i difetti del programma.

Gli anni '80 eliminano i sogni del testing esaustivo, consolidano un patrimonio di riferimento e i risultati acquisiti aprono una nuova visione: il testing come processo complesso dai costi estremamente elevati.

Si prende atto che il testing può rilevare i difetti, ma non può affermare con certezza la loro assenza.

Proseguono infatti gli studi sui criteri di copertura del codice dai test, ma con un approccio completamente diverso da quello esaustivo.

Si sviluppa quindi il concetto di classi di equivalenza, che suddivide l'insieme di tutte le possibili combinazioni di coppie di dati input-output, validi o non validi, in classi.

Tali classi sono dette equivalenti, poiché si prevede lo stesso comportamento del software per ogni elemento all'interno della stessa classe.

Per ogni funzionalità deve essere presente almeno un caso di test per ogni classe di equivalenza, anche se spesso vengono testati: un elemento che sta al centro della classe, uno che sta sul bordo superiore e uno su quello inferiore.

Un'altra conquista di questo periodo è anche quella che svincola il testing dal legame inscindibile con il programma ed i suoi sviluppatori, vedendolo come un qualsiasi processo di produzione indipendente nelle sue fasi di pianificazione, progettazione e implementazione.

Nel 1983 l'Institute for Computer Sciences and Technology of the National Bureau of Standards pubblica le linee guida per il ciclo di vita, la convalida, la verifica e il testing del software [4, 5].

Dopo tale pubblicazione inizia già un nuovo periodo, chiamato "Evaluation-Oriented Period", il periodo orientato alla valutazione.

Nelle linee guida viene descritta una metodologia comprendente l'analisi, la revisione e le attività di test durante il ciclo di vita del software per fornire una valutazione del prodotto.
Questo documento in primo luogo fa emergere due realtà strettamente correlate con il testing del software: la convalida, volta a esaminare se il sistema costruito risolva i problemi che hanno motivato la realizzazione dell'applicativo, espressi sottoforma di requisiti (informali) definiti dall'utente e tradotti nelle specifiche (rigorose e formali) prodotte dallo sviluppatore o dall'analista, e la verifica, che si occupa di controllare il rispetto di tali specifiche durante l'implementazione.
Convalida e verifica mettono in luce nuovi obiettivi del testing, ovvero quelli di ricercare e trovare difetti nella definizione dei requisiti, nell'implementazione e nel design applicativo.
Al tempo stesso, le linee guida esprimono la convinzione che un'attenta scelta di tecniche di convalida, verifica e testing possano aiutare ad assicurare un elevata qualità nello sviluppo e nella manutenzione di un prodotto software, rendendole quindi parte irrinunciabile del ciclo di vita.

2.2.4 – Anni '90

Nel libro "Software Testing Techniques" del 1990, Beizer afferma che "la progettazione di test è una delle più efficaci vie per prevenire gli errori".
Ed è proprio in quello che viene definito "Prevention-Oriented Period", il periodo orientato alla prevenzione, che alla ricerca e alla scoperta di difetti nell'implementazione, nel design e nella definizione dei requisiti di un applicativo, si va ad aggiungere la capacità di poterli prevedere con anticipo.
In questo contesto il meccanismo accresce la sua complessità, si allungano i tempi di realizzazione, con una relativa crescita di costi e risorse dedicati al testing.

Nel 1991 Herzel da la seguente definizione: "il testing è la pianificazione, progettazione, costruzione manutenzione e realizzazione di test ed ambienti di test".

Infatti, in questo periodo si attribuisce un ruolo sempre maggiore alla pianificazione, all'analisi e ai piani di test, alla progettazione e alle strategie di testing da intraprendere.

Proprio per questo una delle sfide di questi anni è l'abbattimento dei costi, che ad oggi pesa dal 40 al 50 percento sul costo di un software.

Ad esempio, sono numerosi i tentativi di automatizzare maggiormente il processo di testing, portando allo sviluppo di diversi tool automatici per il supporto al processo di testing, dalla pianificazione, alla progettazione, all'implementazione dei casi di test.

Si raggiunge finalmente la consapevolezza che il testing, parte integrante e fondamentale all'interno del ciclo di vita del software, sia un vero e proprio processo indipendente per il quale si ha l'esigenza di definire modelli di riferimento.

Proprio per questo motivo iniziano a diffondersi società che offrono la possibilità di eseguire il processo di testing in outsourcing, ovvero affidando l'onere di tale processo all'esterno dell'organizzazione che si è occupata dello sviluppo del software.

Questi sono anche gli anni cruciali per il consolidamento del rapporto tra testing e qualità del software, grazie a una più profonda conoscenza di questi concetti.

Anche il concetto di qualità del software subisce una rapida evoluzione in questi anni, impegnando direttamente o indirettamente gran parte della ricerca nel campo dell'ingegneria del software.

Questo ha permesso di giungere alla conclusione che per qualità del software si intende la misura con cui un applicazione soddisfa un certo

numero di aspettative rispetto al suo funzionamento e alla sua struttura interna.

I parametri con cui possono essere valutate le aspettative di un utente sono ad esempio: affidabilità, correttezza, efficienza ed usabilità, mentre la qualità percepita da uno sviluppatore può essere espressa in termini di: manutenibilità, evolvibilità, portabilità, riusabilità, leggibilità e altro ancora.

I parametri definiti per utenti e sviluppatori sono spesso in relazione tra loro, infatti, banalizzando la questione, si può dire che un software mal scritto tende anche a funzionare male [6].

Quindi, se produrre un software di qualità significa definire in modo appropriato dei requisiti che rispondano alle esigenze dell'utente finale e se tali requisiti devono essere implementati correttamente dalle specifiche definite dallo sviluppatore, allora possiamo anche dire che il testing è quell'attività che ha lo scopo di trovare e prevenire i difetti di qualità di un applicativo.

Una maggiore consapevolezza dell'importanza a livello qualitativo del testing spinge sempre di più per investimenti maggiori in questa fase del ciclo di vita del software.

Negli anni novanta nasce anche un nuovo tipo di procedimento, detto sviluppo "test driven" [7], ovvero guidato dai test, che consiste nel considerare il teting una parte integrante del prodotto software.

Questa tipologia di sviluppo prevede la scrittura di un test ancora prima che esista il codice da testare e ovviamente questo causa un fallimento del test.

Si procede, quindi, alla produzione del codice minimo per cui il test possa essere superato e il software abbia il comportamento desiderato.

Infine, si esegue un raffinamento del codice affinché questo si adegui a standard qualitativi più elevati.

Le motivazioni che hanno portato alla nascita di questa tecnica sono da attribuire principalmente all'equivalenza dei requisiti del prodotto software e dei requisiti considerati nel processo di testing, che vengono definiti prima e determineranno poi sia lo sviluppo del codice, che i casi di test.

Anche l'architettura del software ha una stretta corrispondenza con quella delle procedure di test, infatti è buona norma produrre i casi di test in modo parallelo al codice da testare.

2.3 SCOPO DEL TESTING DEL SOFTWARE

2.3.1 – Difetti e Malfunzionamenti

Se da un lato si può affermare che l'obiettivo principale del testing sia quello di identificare e prevedere quanti più difetti possibili di un'applicazione tramite i suoi malfunzionamenti, per evitare che essi possano verificarsi durante il normale funzionamento, dall'altro è necessario capire la natura di tali difetti e in che modo essi possano generare i malfunzionamenti [10].

I difetti di un prodotto software saranno sempre presenti in una certa misura e ciò non dipende da una disattenzione o irresponsabilità dello sviluppatore, ma perché la complessità del software è generalmente intrattabile e gli uomini hanno capacità limitate per gestire la complessità.

Una prima tipologia di difetti [c, d] che si possono presentare in un'applicazione sono quelli rilevabili da malfunzionamenti del software in fase di esecuzione, solitamente causati da difetti di design applicativo o da difetti di codice.

I difetti di design applicativo evidenziano un'inadeguatezza della struttura del programma, come ad esempio l'errata interazione tra moduli o la scelta sbagliata di un algoritmo.

I difetti di codice sono invece legati ad un uso scorretto del linguaggio di programmazione e possono essere errori di sintassi, dovuti alla mancanza di comprensione del design applicativo o ad un uso sbagliato dei costrutti.

L'esecuzione delle porzioni di codice contenenti tali difetti provocano i malfunzionamenti, che risultano quindi legati in modo diretto.

Un altro tipo di difetti sono quelli più difficili da osservare in fase di esecuzione del software.

In questo caso la percezione di un malfunzionamento si manifesta in modo evidente tramite un comportamento difforme dalle specifiche o dai

requisiti impliciti o espliciti dell'applicazione, ma la causa di tale malfunzionamento non risulta altrettanto chiara.

L'esempio più classico di questo tipo di difetti sono quelli legati all'usabilità o all'aspetto grafico di un applicativo.

La difficoltà nel rilevare tali difetti sta nella soggettività con cui i malfunzionamenti vengono percepiti e la mancanza di relazioni che legano tali percezioni ai difetti oggettivi del prodotto software.

L'ultima tipologia di difetti sono quelli non riscontrabili attraverso i malfunzionamenti.

Se precedentemente abbiamo definito un malfunzionamento come un comportamento difforme dai requisiti e dalle specifiche, un difetto inerente la definizione delle specifiche e dei requisiti stessi non è rilevabile.

Nel caso in cui le specifiche fornite dallo sviluppatore non rispondano completamente alle esigenze dell'utente finale, espresse con i requisiti informali, si possono verificare ambiguità, contraddizioni, imprecisioni e mancanze nella loro definizione.

Lo stesso discorso può essere fatto per i difetti presenti nei casi di test e nella loro progettazione e pianificazione.

Pertanto, in questo contesto, se un test non rileva malfunzionamenti non significa che l'applicazione sia priva di difetti, ma bensì che il difetto potrebbe risiedere nei test stessi o nella definizione dei requisiti e delle specifiche.

2.3.2 – Obiettivi del Testing

Alla luce di quanto emerso riguardo ai difetti di un prodotto software possiamo affermare che l'obiettivo principale del testing è proprio quello di rilevare tali anomalie, di qualsiasi tipo o natura.

Fornendo tramite i casi di test un'adeguata copertura del codice sorgente, ovvero facendo in modo che vengano testate tutte le porzioni di codice, quindi eseguendo tutti i cammini possibili, e analizzando tutte le possibili configurazioni di test secondo criteri di classi di equivalenza è possibile rilevare l'insorgere di tali difetti in fase di esecuzione.
Come abbiamo visto, però, tale operazione potrebbe non essere sufficiente per identificare tutti i tipi di difetti analizzati in precedenza.
In un contesto simile, assumono un ruolo fondamentale anche le fasi di convalida e verifica del software, parti integranti del processo di testing.
La convalida [e], infatti, assicura che si stia costruendo il giusto prodotto, facendo in modo che le esigenze dell'utente finale siano perfettamente rappresentate sia nella definizione dei requisiti informali, che nelle relative specifiche formali, e che l'applicazione faccia ciò che l'utente ha realmente chiesto.
La verifica, invece, assicura che si stia sviluppando il prodotto in modo corretto, quindi che il software sia conforme in ogni sua parte alle specifiche formali.
Un insieme di test ideali in fase di esecuzione, unitamente ad una verifica e una convalida del software perfette, quindi, possono esprimere con alta probabilità la presenza o meno di difetti.
Ciò significa che lo scopo del processo di testing, oltre all'identificazione dei difetti esistenti, può essere anche quello di prevedere con una certa probabilità l'insorgere o meno di eventuali malfunzionamenti, causati da tali difetti, durante la normale operatività del software.
Essendo i malfunzionamenti dei comportamenti dell'applicazione difformi dai requisiti impliciti o espliciti definiti per l'applicazione, possiamo dire che, per un prodotto software testato accuratamente, si conoscono e si possono prevedere le caratteristiche presentate in tali specifiche.

Se le specifiche definiscono correttezza, affidabilità, robustezza, efficienza, usabilità, riusabilità, manutenibilità e portabilità di un prodotto software, allora il testing ci permette di conoscere tali caratteristiche con una precisione proporzionale alla probabilità dell'esistenza di difetti, poiché esse dipendono direttamente o indirettamente dal manifestarsi di malfunzionamenti.

Detto ciò, se il concetto di qualità, come definito nello standard [11] è la "capacità di un insieme di caratteristiche inerenti ad un prodotto, sistema, o processo di ottemperare a requisiti di clienti e di altre parti interessate", possiamo giungere alla conclusione che un altro scopo del testing è proprio quello di misurare e prevedere un certo grado di qualità dell'applicativo realizzato.

2.4 TECNICHE DI TESTING DEL SOFTWARE

2.4.1 – Analisi Statica

L'analisi statica è il processo di valutazione di un sistema o di un suo componente basato sulla sua forma, sulla sua struttura, sul suo contenuto o sulla documentazione di riferimento [12].

Ciò significa che la valutazione avviene a prescindere dall'esecuzione del sistema o dell'oggetto che si sta testando.

Nonostante con il termine testing generalmente non ci si riferisca a questa tecnica, l'analisi statica è solitamente il primo controllo effettuato sul codice e sui requisiti applicativi con l'intento di riscontrare anomalie nel prodotto software.

Alcuni esempi di analisi statica sono: la compilazione del codice, revisioni, ispezioni, recensioni, analisi del flusso dei dati e analisi del flusso di controllo.

I compilatori eseguono un analisi statica, per verificare che un programma soddisfi particolari caratteristiche di correttezza per poter generare il codice.

Le anomalie rilevate dal compilatore all'interno del codice possono variare in base al linguaggio di programmazione, ma in genere possono essere identificati: nomi di identificatori non dichiarati, incoerenza tra tipi di dati coinvolti in una istruzione e codice non raggiungibile dal flusso di controllo.

Un altro tipo di analisi statica è il "code reading", ovvero l'attenta rilettura del codice da parte del programmatore stesso o di un'altra persona.

Questa operazione può portare alla luce nuovi difetti, quali ad esempio: loop infiniti, inefficienza di algoritmi e non strutturazione del codice.

Quando quest'attività viene effettuata da un gruppo di persone preposte a tale mansione, tra cui solitamente almeno uno dei programmatori, e

successivamente discussa e approfondita, viene anche chiamata ispezione o revisione.

L'analisi del flusso di controllo è invece una tecnica con cui viene data una rappresentazione del codice in un grafo, dove i nodi denotano istruzioni o predicati, mentre gli archi il passaggio del flusso di controllo.

L'attento esame di tale grafo può evidenziare l'esistenza di eventuali anomalie quali codice irraggiungibile e non strutturazione.

Infine, l'analisi del flusso dei dati esamina l'evoluzione del valore delle variabili durante l'esecuzione di un programma in modo statico, ovvero considerando le operazioni effettuate su tali variabili e non i dati veri e propri in esse contenuti in fase di esecuzione.

E' importante associare all'analisi statica, anche il rilevamento di difetti generati in tutte quelle operazioni di analisi e progettazione dei requisiti/specifiche per l'applicazione che si intende realizzare.

I difetti che possono presentarsi sia per i requisiti informali, definiti dall'utente finale, che per le specifiche formali, definite dallo sviluppatore o dall'analista, sono ad esempio: ambiguità, contraddittorietà, imprecisione, mancanza.

L'analisi statica dei requisiti e delle specifiche, specialmente se effettuata prima dello sviluppo vero e proprio, può riscontrare un difetto alla radice, riducendo la probabilità di generarne altri e con un conseguente risparmio di tempo, risorse e denaro.

2.4.2 – Analisi Dinamica

Tipicamente con il termine testing ci si riferisce all'analisi dinamica, che è il processo di valutazione di un sistema software o di un suo componente basato sull'osservazione del suo comportamento in esecuzione.

La precondizione necessaria per poter effettuare un test è la conoscenza del comportamento atteso per poterlo confrontare con quello osservato.

L'oracolo è quella componente che conosce il comportamento atteso per ogni caso di test e può essere umano, quindi basato sulla conoscenza delle specifiche e sul giudizio personale, oppure automatico, generato dalla definizione di specifiche formali oppure basato sui risultati di un software dello stesso tipo, ma sviluppato da altri.

Altri dati necessari per un'analisi dinamica sono una corretta configurazione del software, che include la specifica dei requisiti, le caratteristiche grafiche e di usabilità desiderate, la struttura dell'implementazione ed il codice sorgente, e la configurazione dei test, che comprende tutta la documentazione relativa al piano di test, le procedure e i casi di test e i tool specifici per il testing.

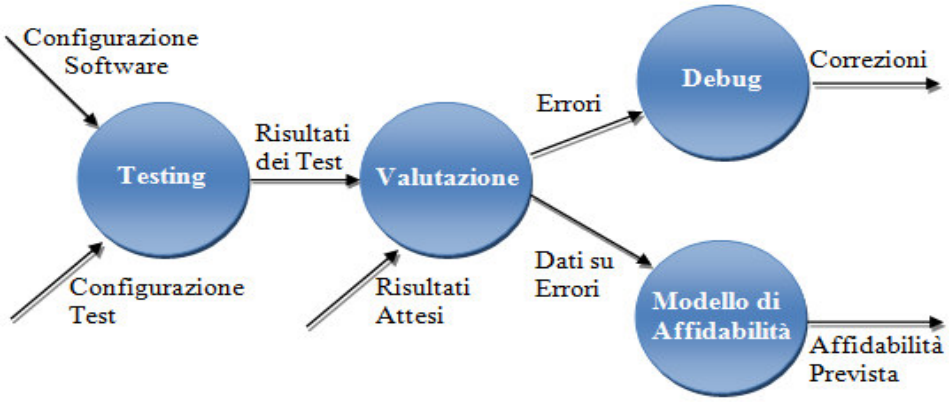

Figura 1 - *Flusso delle informazioni nel processo di testing.*

Come mostrato in figura la prima fase di un'analisi dinamica prevede l'esecuzione di tutte le procedure e dei casi di test, prendendo come input sia la configurazione del software che quella dei test.

I risultati prodotti verranno presi in input dalla seconda fase, che insieme ai valori attesi elaborati da un oracolo darà vita alla vera e propria fase di valutazione.

Questa fase prende solitamente il nome di convalida o verifica, in base al tipo di valutazione eseguito.

La convalida ha lo scopo di esaminare se il sistema costruito risolva i problemi che hanno motivato la realizzazione dell'applicativo, espressi sottoforma di requisiti (informali) definiti dall'utente, mentre la verifica, si occupa di controllare il rispetto delle specifiche (rigorose e formali) prodotte dall'analista a partire dai requisiti iniziali.

La valutazione produce due tipi di risultati: gli errori riscontrati, che verranno dati in input al processo di debug, con lo scopo di correggere tali anomalie, ed una serie di dati sugli indici di errore, che saranno analizzati in un modello di affidabilità, per determinare il grado di affidabilità previsto per l'applicazione.

Una prima classificazione di analisi dinamiche avviene in base all'approccio adottato: il "black box testing" o testing funzionale e il "white box testing" o testing strutturale [13].

Mentre il testing funzionale è fondato sull'analisi degli output generati dal sistema o dai suoi componenti in risposta ad input definiti sulla base della sola conoscenza dei requisiti di sistema specificati, il testing strutturale richiede un'approfondita conoscenza della struttura del software e del suo codice, a partire dalla quale vengono definiti dei casi di test, gli input ad essi associati ed i risultati attesi.

Il testing funzionale mette in evidenza il comportamento esterno del software, senza sapere cosa accade all'interno di esso, da qui appunto prende il nome di "black box" (scatola nera), alla quale viene passato un determinato input e ne viene estratto un certo output.

Questo tipo di test viene effettuato accedendo al software solamente tramite l'interfaccia utente, oppure tramite interfacce di comunicazione tra processi.

A causa della natura di tale tecnica appare evidente sin da subito che non si conosca la parte del codice che effettivamente si stia testando, con il relativo svantaggio di lasciare alcune parti di codice senza test e altre con

test ridondanti, ma con il vantaggio di assicurare la qualità di tutte le funzionalità dell'applicativo in modo indipendente dall'implementazione.
Un altro vantaggio del testing a scatola nera sta nel fatto che tale test può essere effettuato anche da persone esterne all'azienda.
Infatti, non sono necessari programmatori esperti e che conoscano gli aspetti interni del software da collaudare.
Pertanto, si possono trovare molti più collaudatori, senza dover investire nell'addestramento.
Tale vantaggio si manifesta in misura ancora maggiore se il software da testare, il codice sorgente e la relativa documentazione sono coperti da segreto industriale.
Il testing strutturale, invece, il software viene visto come una "white box" (scatola bianca), per contrapposizione alla tecnica a scatola nera.
In questa metodologia è richiesta una conoscenza approfondita del codice sorgente, degli algoritmi utilizzati e della struttura interna del software.
Tipicamente il testing strutturale viene eseguito per i test di singoli moduli e per i test delle singole unità in modo automatizzato, ovvero prevedendo procedure e casi di test che invocano procedure vere e proprie dell'applicazione e raccolgono dati durante la loro esecuzione.
I test a scatola bianca possono essere scritti nello stesso linguaggio di programmazione dell'applicativo, oppure con un linguaggio in grado di interfacciarsi con quest'ultimo.
L'obiettivo della selezione dei casi di test e dei dati di input è quello di causare l'esecuzione di specifici punti del software.
Le tecniche di testing strutturale sono in genere fondate sull'adozione di metodi di copertura di tali punti: istruzioni, procedure e porzioni di codice che compongono la struttura dei programmi.
Per copertura si intende la definizione di un insieme di casi di test, ed in particolare dati di input, in modo tale che tutti gli oggetti che compongono

la struttura del programma applicativo siano attivati almeno una volta nell'esecuzione dei casi di test.

Il collaudo a scatola bianca, siccome richiede l'interfacciamento con le singole procedure, può essere effettuato solamente da un programmatore, anche se non necessariamente un membro del gruppo che ha sviluppato il software da collaudare.

Normalmente è effettuato comunque da un membro dell'organizzazione che sviluppa il software.

Un vantaggio del collaudo a scatola bianca è che permette di sondare dettagliatamente tutte le casistiche che possono presentarsi in modo facile ed efficiente grazie all'automazione.

Un altro vantaggio è che è più facile il debugging, cioè passare dal malfunzionamento alla correzione del difetto, in quanto la segnalazione del malfunzionamento normalmente indica il punto del codice e i valori delle variabili per cui il malfunzionamento si è manifestato.

2.5 TIPOLOGIE DI TESTING DEL SOFTWARE

2.5.1 – Test di Unità

Lo scopo dei test di unità [14] è quello di isolare ciascuna parte di un programma e mostrarne correttezza e completezza nell'implementazione, facendo emergere tempestivamente eventuali difetti, in modo che essi possano essere corretti facilmente prima dell'integrazione.

Il testing di unità è realizzato mediante un approccio white-box testing e si articola tipicamente in "test case", ovvero casi di test, ciascuno dei quali dovrebbe essere indipendente dagli altri.

Dato il forte legame tra codice sorgente, casi di test e relative specifiche da verificare, il test di unità viene normalmente eseguito dallo sviluppatore che si è occupato di quella particolare porzione di programma.

I casi di test, infatti, vengono abitualmente sviluppati in parallelo al codice applicativo e sono implementati tramite procedure automatiche, in grado di rieseguire il test ogni volta che ce ne sia la necessità.

Il motivo per cui tale tipologia di testing risulta fondamentale e irrinunciabile è che spesso i difetti sono più facilmente identificabili tramite i malfunzionamenti avvenuti in un ambiente più piccolo e circoscritto.

Se i difetti vengono identificati e risolti in breve tempo dopo la loro introduzione, il tempo per la loro localizzazione e soluzione risulta notevolmente ridotto.

Il compito del testing di un intero applicativo risulta notevolmente semplificato si è già testata la correttezza di ogni singolo modulo.

Un'altra buona ragione per eseguire i test di unità è che alcuni difetti emergono solamente in presenza di particolari configurazioni del software, che sarebbero difficilmente ricreabili in un ambiente più complesso e vasto.

Spesso vengono introdotti a tal proposito degli appositi "driver", che simulano le unità chiamanti della componente testata, e degli "stub", che simulano l'unità chiamata.
Inoltre, il test di unità abbassa i costi, in termini di tempo e risorse, per l'identificazione e la correzione di difetti, rispetto a quelli che si sarebbero dovuti affrontare per ottenere lo stesso risultato tramite dei test sull'intero applicativo.
Nella programmazione orientata agli oggetti la tipica componente che costituisce l'unità da testare individualmente è la classe, ed il test di unità più piccolo include il costruttore ed il distruttore.
In generale il testing non riesce ad identificare tutti gli errori in un programma e lo stesso vale per i test di unità che, analizzando per definizione le singole componenti, non possono rilevare difetti di integrazione, problemi legati alla performance e altri problemi legati al sistema in generale.
Il test di unità risulta quindi più efficace se utilizzato in congiunzione con altre tecniche di testing del software.

2.5.2 – Test di Integrazione

I test di integrazione sono definiti come livello intermedio di testing e abitualmente seguono a quelli di unità e precedono quelli di sistema.
Infatti, tali test vengono eseguiti quando due o più unità già testate vengono aggregate in una struttura più grande, rappresentando l'estensione logica del test di unità.
Il testing delle parti combinate di un applicazione ha il principale scopo di determinare se esse hanno il comportamento atteso quando si trovano ad operare insieme come un unico componente.

Solitamente vengono previsti test su tutte le interfacce tra i singoli componenti, in modo da estendere la valutazione sul grado di qualità dai singoli moduli alla struttura più grande considerata.
Per una più facile identificazione di difetti di integrazione questo tipo di testing viene attuato aggiungendo in modo progressivo un componente alla volta all'insieme di quelli che sono già stati testati.
Per risalire al problema che ha generato il malfunzionamento basterà quindi risalire all'ultimo modulo incluso nel test di integrazione.
Per tale ragione questa tipologia di testing richiede una particolare cura nella documentazione, nella raccolta e nella documentazione dei test effettuati.
Anche questa tipologia, come visto anche nei test di unità, utilizza principalmente un approccio di tipo white box testing, poiché si prende in esame il software dall'interno e si rende necessaria un'approfondita conoscenza della struttura interna, per poter operare scelte coerenti sulla progressiva integrazione di ciascun modulo.
In tal senso, le due strategie più utilizzate sono di tipo "top-down" oppure "bottom-up", dove nel primo caso viene prima testata la corretta integrazione delle funzionalità di alto livello, passando poi progressivamente alle procedure di livello più basso da queste richiamate, viceversa nel secondo caso.

2.5.3 – Test di Sistema

Dopo aver completato i test sulle singole unità e verificato la loro corretta integrazione, si procede con il collaudo dell'intero sistema.
Effettuare dei test sull'intero sistema significa affermare con una certa misura la qualità del prodotto finale.
Le caratteristiche testate a questo livello possono essere diverse e numerose, e spesso dipendono dal tipo di applicazione realizzata.

Tali test solitamente si basano sulle funzionalità espresse nelle specifiche e nei requisiti dell'applicazione e possono coinvolgere diverse configurazioni software e hardware.

Anche se l'approccio utilizzato è per lo più di tipo black box testing, quindi orientato ai test funzionali, vengono controllate anche altre caratteristiche strutturali come sicurezza, usabilità e manutenibilità.

I test di sicurezza [15], ad esempio, sono un aspetto cruciale del collaudo di un sistema, e dipendono necessariamente dai requisiti definiti per l'applicazione che si sta realizzando, che deve poter garantire caratteristiche come: autenticità, controllo degli accessi, segretezza dei dati, integrità dei dati e non ripudiabilità.

Il testing di sicurezza può risultare molto impegnativo, complicato e costoso per applicazioni che richiedono un elevato grado di sicurezza, in quanto per proteggere il software da eventuali attacchi bisogna tenere conto che questi sono generalmente provocati da avversari intelligenti che sfruttano tutte le possibili debolezze del sistema.

Il testing accurato dei meccanismi di sicurezza, sviluppati per prevenire, scoprire o ripristinare il sistema dopo un attacco, viene sempre affiancato alle analisi di rischio, che possono aiutare a identificare e classificare potenziali problemi di sicurezza e il loro impatto.

Mentre le analisi di rischio vengono affrontate con un approccio di tipo white box testing, attraverso una profonda conoscenza dell'architettura del software e una ricerca e classificazione delle sue debolezze interne, il testing dei meccanismi di sicurezza, invece, viene solitamente effettuato con test funzionali di tipo black box.

Un altro classico test di sistema, oltre a quelli legati alla sicurezza, è il cosiddetto "stress test" [e], che ha lo scopo di destabilizzare il sistema ponendolo in uno stato di stress, quindi con un sovraccarico o una sottrazione di risorse.

Questo tipo di testing ha come scopo quello di garantire l'aspetto qualitativo noto come "recoverability" o robustezza, ovvero la capacità di un sistema di reagire ad errori imprevisti.

Dopo aver mandato in crash il sistema è necessario osservare con attenzione la sua reazione, misurando la sua tolleranza verso errori improvvisi, verificando se l'arresto avviene in modo controllato o entra immediatamente in uno stato di blocco, controllando se il riavvio riprende dall'ultimo stato corretto e se i dati sono stati salvati o andati persi.

Alcuni esempi di stress test per un applicazione web possono essere: lo spegnimento e il riavvio casuale di porte di switch e router di rete che collegano i server, la messa offline del database e l'esecuzione di processi che consumano risorse (CPU, memoria, disco e rete) sui server.

I requisiti di un applicazione comprendono anche l'efficienza del prodotto software, tipicamente controllata nei test di sistema.

Lo scopo di questo tipo di test è proprio quello di andare a verificare che siano soddisfatti quelli che vengono comunemente definiti requisiti prestazionali, che devono quindi essere verificati prima del rilascio dell'applicativo.

Questa tipologia di test viene anche spesso utilizzata per il confronto in termini di efficienza di diverse versioni di uno stesso applicativo.

Solitamente in questo tipo di test la variabile principale che viene considerata è il tempo di esecuzione.

Infatti, per vari tipi di operazioni vengono fissati i tempi massimi di esecuzione (ovvero, si definiscono delle "baseline") e si verifica che il prodotto software non superi tali tempi limite.

I tempi massimi di esecuzione presi in esame possono variare in base al tipo di hardware e alle funzioni del software analizzate.

E' chiaro che se tale test fosse eseguito per un applicazione web installata su server di ultima generazione, con tecnologia multiprocessore, i risultati

attesi sarebbero molto diversi da quelli della stessa applicazione su un server obsoleto.

Lo stesso discorso può essere fatto per la complessità computazionale delle varie operazioni prese in esame, che determineranno in modo sostanziale tali limiti di tempo.

Questi test possono essere sviluppati sia dal punto di vista funzionale, che da quello strutturale, ovvero ispezionando e controllando il sistema dall'interno, considerando i tempi di esecuzione come somma dei tempi necessari per le singole operazioni.

Ciò si spiega con la raccolta e l'analisi di misure localizzate in diverse porzioni di codice, che facilitano la determinazione di un rallentamento dell'applicazione.

Uno o più test di performance possono essere integrati anche nei test di regressione, per verificare che le modifiche al prodotto software non abbiano introdotto rallentamenti.

Un altro tipo di test di sistema è il "load test", che prevedel'aumento costante del carico sul sistema tramite strumenti automatici.

Per una applicazione web, ad esempio, il carico è il numero di utenti/connessioni HTTP concorrenti.

Altri esempi di questo tipo di test possono essere: la creazione di un grande numero di mailbox in un mail server, la scrittura di un documento molto lungo per un word processor oppure la stampa di un job pesante per una stampante.

Questo tipo di test è molto utile per prevenire difetti legati alla gestione della memoria o alle dimensioni di buffer.

2.5.4 – Test di Accettazione

I test di accettazione [f] vengono eseguiti immediatamente prima del rilascio del prodotto, quindi dopo aver testato l'intero sistema e dopo il conseguente "bug fixing", ovvero la correzione dei difetti riscontrati.
Sono tipicamente di approccio black box testing, quindi orientati alla verifica dal punto di vista strettamente funzionale dell'intero sistema e vengono chiamati anche test di qualità, test finali o accettazione di rilascio.
Proprio dal nome di tali test si evince che abbiano più lo scopo di dare fiducia sull'adeguato funzionamento del sistema, piuttosto che quello di trovare difetti.
Infatti, il termine accettazione sta proprio a indicare che tramite questi test si vuole stabilire se il prodotto finito possa essere accettato come soluzione alle reali necessità dell'utente finale.
Il testing di accettazione generalmente comporta l'esecuzione di una collezione di casi di test, ognuno dei quali esercita una particolare funzionalità operativa del sistema, restituendo come risultato il successo o l'insuccesso di tale operazione.
Generalmente non ci sono gradi di successo o di insuccesso per questo tipo di casi di test, ma solamente l'uno o l'altro risultato.
Ciascun caso di test viene solitamente eseguito in un ambiente identico, o il più simile possibile, a quello reale, in cui si troverà ad operare l'utente finale, e deve essere accompagnato dai relativi dati di input o da una descrizione formale delle attività operazionali per essere eseguito e da una descrizione formale dei risultati attesi.
I test di accettazione possono essere eseguiti sia da chi ha sviluppato il sistema, che dagli utenti finali prima di entrare in possesso del software.
Nel caso in cui i test vengano effettuati direttamente dagli utenti finali, si parla di "User Acceptance Testing", che ha come obbiettivo quello di

confermare che il sistema sia conforme ai requisiti reciprocamente concordati tra le parti.

I test eseguiti dall'utente finale o dal cliente, oltre ad essere una conferma della misura di correttezza dal punto di vista funzionale dell'applicativo, spesso hanno anche una valenza legale da un punto di vista contrattuale, come momento conclusivo della vendita o del rilascio del software.

Altri due esempi di test di accettazione sono l'alfa e il beta testing, anche essi eseguiti sull'intero applicativo, in un ambiente di test identico o molto simile a quello reale in cui si troverà normalmente ad operare il software.

L'alfa testing può essere eseguito da potenziali utenti finali o da sviluppatori interni o esterni al team di sviluppo, ma avviene sempre internamente all'organizzazione che si è occupata della costruzione del software.

Proprio per questo motivo a volte viene usato il nome di test di accettazione interno.

Spesso dopo tali test segue un raffinamento del software, quindi non è completamente assente la componente dell'identificazione dei difetti, anche se abitualmente marginali o comunque non così gravi da minacciare le funzionalità principali.

Questi test precedono il passaggio al beta testing, caratterizzato dal rilascio di una o più versioni beta del prodotto al di fuori dell'organizzazione che ha sviluppato il software.

Una versione beta è una versione di prova del software, o comunque non definitiva, solitamente testata da un gruppo selezionato di utenti finali o anche da tutti loro.

Tali versioni sono tendenzialmente simili al prodotto software finale, ma vengono distribuite senza alcuna garanzia e possono essere instabili.

Con lo sviluppo di internet il beta testing si è diffuso enormemente, soprattutto per i prodotti software commerciali, che solitamente vengono

distribuiti in rete gratuitamente in versione beta, in modo che possa essere provato da un gran numero di utenti con configurazioni hardware e software differenti.
Una volta riscontrato un difetto o un problema di compatibilità, viene chiesto all'utente di inviare una segnalazione al produttore del software.
Quando la frequenza delle segnalazioni si abbassa, fino ad un livello accettabile, è giunto il momento del rilascio della versione definitiva del prodotto software, detta anche "build version".

2.5.5 – Test di Regressione

Questa tipologia di test ha lo scopo di verificare la qualità di nuove versioni di un prodotto, in particolare l'intento è quello di testare le nuove funzionalità e garantire che le funzionalità preesistenti abbiano mantenuto le loro caratteristiche qualitative dopo l'introduzione di queste ultime.
Proprio da qui il nome di test di regressione, poiché ha proprio lo scopo di controllare se la qualità del prodotto sia regredita.
Dopo l'introduzione di nuove componenti o funzionalità, per assicurare il livello di qualità, sarebbe necessario ripetere l'intero processo di testing su tutto l'applicativo, ad eccezione dei test di unità già effettuati.
Proprio per evitare di dover far fronte nuovamente agli alti costi di tale processo, i test di regressione vengono pensati in anticipo, predisponendo la reiterazione dei test in modo automatico per le nuove versioni del prodotto, a seguito di modifiche marginali sull'intera collezione dei test.

3.

CAPITOLO TERZO

LA DOCUMENTAZIONE NEL PROCESSO DI TESTING

Il processo di testing si sviluppa in varie fasi, ad ognuna delle quali è associata una ricca produzione e consultazione di documentazione di vario tipo.

Alcuni di questi documenti a volte vengono redatti, per la volontà di coloro che si occupano del testing, perché il collaudo possa avvenire in modo formale e rigoroso, ma nella maggior parte dei casi risulta indispensabile e necessaria per una buona riuscita del processo.

In questo capitolo vengono descritte nel dettaglio le varie fasi che caratterizzano il processo di testing tradizionale, mettendo in evidenza la

stretta relazione che hanno con la documentazione di test da esse prodotta.
Verrà poi data una visione complessiva di tali documenti, mettendo in evidenza i rapporti che li legano in un'unica struttura organizzata.
Infine, si arriverà a definire la documentazione in base al ruolo ricoperto all'interno del processo di testing e l'importanza della sua presenza [16].
Per finire, verranno delineate alcune difficoltà legate alla produzione e aggiornamento della documentazione di test per progetti Open Source.

3.1 IL PROCESSO DI TESTING

3.1.1 – Pianificazione dei Test

Il processo di testing del software viene avviato con la fase di pianificazione delle attività di test, che solitamente è orientata alla produzione di un documento chiamato "Piano di Test".
Il piano di test solitamente riguarda tutte le tipologie di testing che si intendono sviluppare per l'applicativo.
In tale documento si fornisce una descrizione e la tipologia dell'applicativo, le funzionalità e i moduli da sottoporre ai test e quelli per cui non è previsto, le tipologie di testing da eseguire, l'ordine di realizzazione e il livello di priorità dei test, gli ambienti di test e le risorse hardware e software necessarie alla loro realizzazione.
La fase di pianificazione incomincia con la raccolta dei requisiti dell'applicazione e delle specifiche di progetto, utili a determinare l'approccio generale al testing e le risorse necessarie, come ad esempio, nel caso dei test di unità, l'adozione di particolari framework o altro software.

Si arriva poi a produrre quello che viene comunemente chiamato schedule generale, ovvero una lista di attività di testing in un certo ordine temporale e con un indice di priorità associato ad ognuna.
La determinazione dello schedule considera le caratteristiche da testare e le relative tipologie di testing associate, come ad esempio: test di accettazione degli utenti, test di sicurezza, stress test, test prestazionali, altri test di sistema e test di integrazione.
I test di unità, in questa fase, vengono solitamente presentati attraverso la suddivisione in moduli applicativi, senza entrare nel dettaglio, ma allo stesso tempo cercando di definirne gli ambienti di test e le caratteristiche generali dei dati in input e output ad esse associati.
Mentre per le altre tipologie di testing, i documenti da cui si estraggono e si elaborano le informazioni per definire il piano di test, sono prevalentemente i requisiti funzionali, per i test di unità vengono usati ampiamente anche le specifiche architetturali del progetto.
Dopo aver richiesto eventuali chiarimenti sui requisiti, si passa al raffinamento delle informazioni ricavate da questa fase del processo di testing.
I dati acquisiti possono quindi essere quindi inseriti in modo dettagliato nel piano di test.

3.1.2 – Progettazione dei Test

Questa fase segue alla pianificazione dei test, perciò i principali documenti e informazioni utilizzate provengono dal piano di test, ma anche dalla definizione dei requisiti e dalla specifiche formali dell'applicazione.
Lo scopo della progettazione è quello di acquisire tutti i dati necessari per poter procedere all'implementazione dei test.
Per fare ciò è possibile procedere in due modi: ampliare il piano di test esistente, oppure produrre altri documenti in relazione con esso.

Tali documenti, sia nel caso in cui si decida di mantenerli separati oppure di integrarli nel piano di test, sono: le specifiche dei test, le specifiche dei casi di test e le procedure di esecuzione.
Le specifiche dei test si basano su un ulteriore frazionamento in gruppi delle attività di testing già previste nello schedule del piano di test.
Ogni raggruppamento indica una tipologia o un particolare test operato, e ne viene descritto l'ambiente di test ed un elenco di casi di test appartenenti, completi di attributi generali e motivazione.
La specifica di un caso di test, invece, è un documento che identifica una particolare configurazione di test, descrivendone gli input inseriti, i risultati attesi, le condizioni di esecuzione, l'ambiente hardware e software.
Infine, un documento relativo alle procedure di esecuzione può essere associato ad una specifica di test o ad uno o più casi di test, e descrive i passi da fare per eseguirli.
In questo tipo di documenti può essere specificato come avviare la procedura di test, come effettuare le misure e rilevazioni, gli eventi imprevisti che possono accadere durante l'esecuzione e come comportarsi nel caso in cui si verifichino.

3.1.3 – Implementazione dei Test

L'insieme dei documenti precedentemente descritti costituisce l'input necessario per la fase di implementazione dei test, che porterà alla realizzazione vera e propria di quanto emerso nella fase di progettazione.
Proprio in questa fase, infatti, vengono valutati i singoli casi di test e implementati come previsto nelle specifiche.
Solitamente per fare questo vengono create delle procedure in grado di eseguire automaticamente i test, in modo da poterli eseguire ogni qual volta se ne presenti la necessità.

Tali procedure devono verificare la presenza di condizioni che siano in grado di stabilire il superamento con successo del test oppure un suo fallimento, in base al confronto tra il risultato atteso per il caso di test ed il valore effettivamente ottenuto dall'applicazione.

L'introduzione di procedure automatiche permette anche un alto grado di riusabilità dei test, che, a seguito di qualsiasi cambiamento del prodotto, dovranno solo essere modificati in maniera marginale.

E' il caso, ad esempio, dei test di regressione.

L'implementazione tiene conto anche dei documenti relativi alle procedure di esecuzione precedentemente definite, che indicano il modo con cui devono essere eseguiti i test.

Tali documenti possono ovviamente essere modificati durante questa fase, nel caso in cui sorgano complicazioni nella realizzazione delle procedure oppure nel momento in cui si identifichino soluzioni più appropriate.

Al termine dell'implementazione, i singoli test realizzati vengono spesso inseriti all'interno di una o più "test suite", ovvero collezioni di test.

Una test suite è una struttura in grado di lanciare l'esecuzione di tutti i test in essa contenuti, senza doverli richiamare uno ad uno.

Assegnare gruppi di test omogenei alla stessa test suite offre anche la possibilità di strutturare e ordinare i test in maniera più efficiente.

Una volta implementate le test suite questa fase può ritenersi conclusa, ma al termine di essa viene solitamente generata la documentazione tecnica del codice realizzato.

Tale documento, comunemente chiamato "API specifications" (Application Programming Interface), presenta mediante un'astrazione l'insieme delle procedure create dallo sviluppatore.

Questo tipo di documentazione risulta particolarmente importante per capire il funzionamento e la struttura dei casi di test, per comprendere

eventuali errori ed eccezioni sollevate durante la loro esecuzione, per l'analisi dei risultati ottenuti e nel processo di debugging.
Infatti, al termine della fase di implementazione i test verranno eseguiti, valutati ed analizzati nel dettaglio.

3.1.4 – Esecuzione e Valutazione dei Test

Al termine dell'implementazione e dopo aver allestito le configurazioni e gli ambienti di test necessari, possono finalmente essere eseguiti i test.
Solitamente è sufficiente eseguire una o più test suite contenenti tutti i test implementati per l'applicativo.
Ogni caso di test lanciato può portare a tre risultati: il superamento, il fallimento, oppure un errore durante l'esecuzione del test.
Quando tutti le condizioni specificate per il superamento di un caso di test, indicano che per ogni input fornito sono stati ottenuti esattamente i risultati attesi, il test risulterà superato con successo.
Quando, invece, una o più condizioni identificano valori ottenuti diversi da quelli attesi, il test rileva un fallimento e indica che probabilmente è presente un difetto nel codice, che viene opportunamente segnalato.
Indipendentemente dai risultati ottenuti dalla valutazione delle condizioni fino a quel momento, quando si presenta un evento imprevisto nell'oggetto del test o nel test stesso, viene segnalata una eccezione non gestita e il test restituisce errore come risultato.
L'esecuzione dei test genera in primo luogo il cosidetto "Log", ovvero una raccolta strutturata di tutti i dettagli inerenti l'esecuzione dei test in ordine cronologico, come ad esempio: il nome del test eseguito, il tempo di esecuzione, data e ora di inizio e il risultato ottenuto.
Studiando poi il test log e incrociando i risultati ottenuti con i requisiti di completezza e terminazione, definiti in fase di pianificazione e progettazione, è possibile eseguire il "check dei test".

Questa operazione di verifica vuole stabilire se tutte le funzionalità siano state testate e se la totalità dei test siano sufficienti per coprire l'intero codice e le diverse classi di input identificati in precedenza.
In generale il check effettua una stima del grado con cui i test siano esaustivi rispetto agli obbiettivi previsti nel piano di test.
Nel caso in cui venga riscontrata la mancanza di test o la ridondanza di alcuni di essi, si provvede a effettuare eventuali modifiche all'insieme dei test esistenti.
Una volta eseguito l'insieme completo dei test per l'applicativo considerato si procede quindi verso la valutazione dei risultati ottenuti.
La prima cosa da fare è quella di salvare i log dei test eseguiti in appositi repository o database.
La memorizzazione persistente di tali informazioni è importante per vedere l'evoluzione di un prodotto e risulta fondamentale nella valutazione di diverse versioni nei test di regressione.
Secondariamente è necessario effettuare una prima analisi dei difetti riscontrati nell'applicativo, che solitamente avviene durante la creazione del "Test Summary Report", ovvero un insieme di valutazioni rispetto ai fallimenti o gli errori riscontrati nei test, che necessitano di ulteriori approfondimenti.
Per i test di unità questo documento costituisce solitamente l'input per il processo di debugging dell'applicativo e contiene il log dei test che evidenziano difetti o che sono andati in errore, delle possibili cause o soluzioni e i limiti riscontrati nei test.
Dopo il processo di debugging, l'esecuzione e la valutazione dei test possono essere reiterati più volte, per accertare il corretto funzionamento delle unità testate.

3.3 RUOLO DELLA DOCUMENTAZIONE

3.2.1 – Correlazione dei Documenti di Test

Il processo di testing, come mostrato in precedenza, genera un quantitativo impressionante di documentazione di vario tipo.

Ognuno di questi documenti è in relazione con gli altri e tutti insieme formano una sorta di reticolo informativo, che deve guidare coloro che si occupano del testing nella pianificazione, nella progettazione, nell'implementazione, nell'esecuzione, nella valutazione e nell'analisi.

Il sistema della documentazione di test è abbastanza complesso e articolato, ma essendo generato da un processo a fasi sequenziali, dove l'output generato dalla fase precedente generalmente costituisce l'input di quella successiva, è possibile ricostruirlo e schematizzarlo.

Come detto in precedenza, un piano di test generale prende corpo essenzialmente dall'analisi dei requisiti e dalle specifiche di progetto nella fase di pianificazione, per essere poi raffinato e integrato con le specifiche dei test da effettuare, con i particolari casi di test e con le procedure per eseguirli.

L'implementazione genera il codice di tutti i casi di test previsti in fase di progettazione, ma anche in questa fase la descrizione di tale codice viene codificata nella documentazione tecnica.

Infine, una volta mandati in esecuzione i test, i risultati prodotti dall'applicazione stessa vengono confrontati con quelli attesi.

I risultati di tale valutazione e analisi vengono salvati in test log e test summary report.

La [Figura 2] mostra come ogni fase del processo di testing venga meticolosamente documentata.

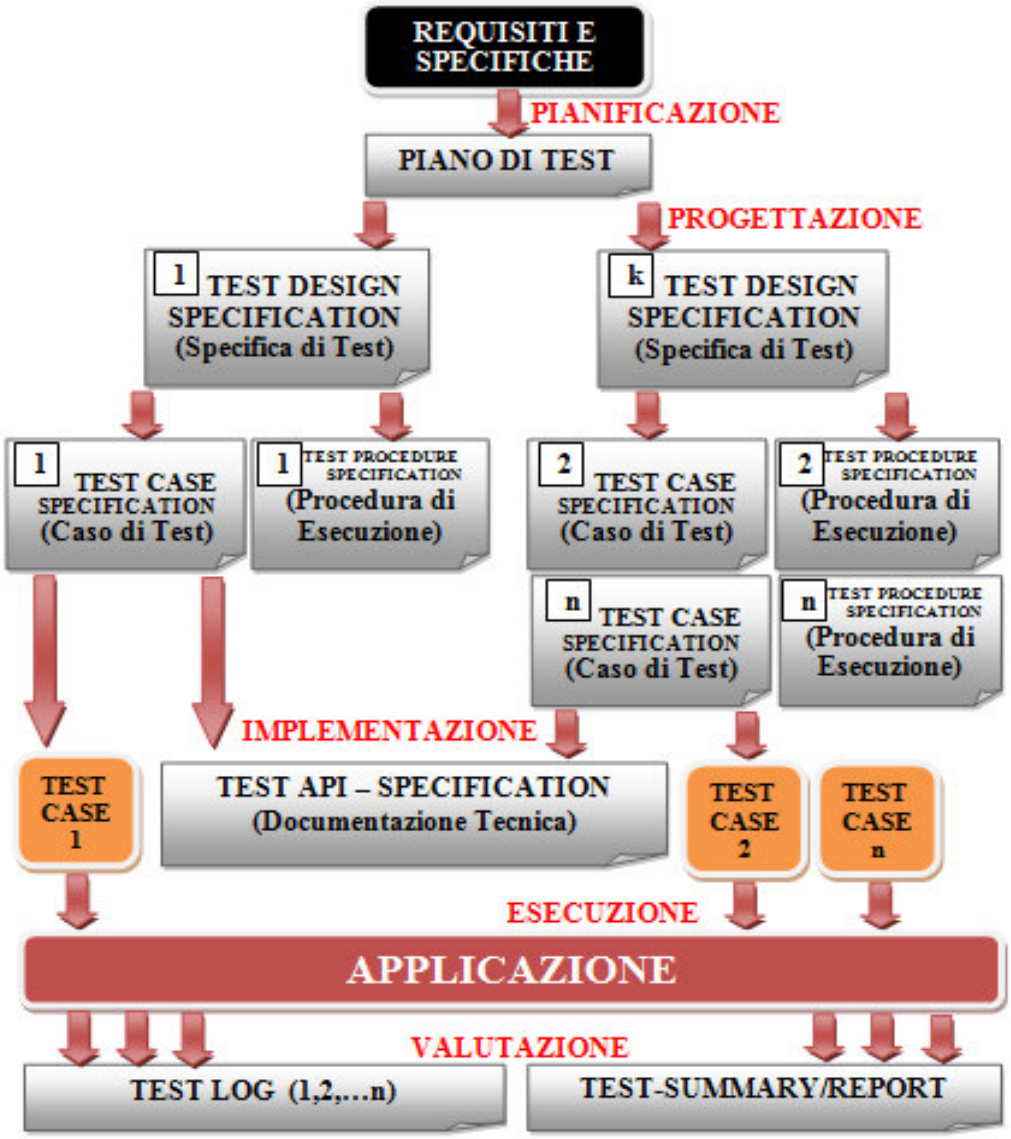

Figura 2 - *La documentazione generata durante il processo di testing.*

3.2.2 – Divario tra requisiti e realtà applicativa

Come è possibile osservare in [Figura 2], l'evolvere del processo di testing tende a mostrare un grado di corrispondenza tra il modello creato sulla base dei requisiti, che rappresenta i risultati attesi, e l'applicativo effettivamente realizzato, che viceversa rappresenta i risultati ottenuti.

Per assurdo, se ciascun test case contenesse un certo numero di condizioni sempre positive, in modo da stabilire l'uguaglianza dei dati ricavati dai requisiti, con quelli ricavati dall'applicazione, allora [Figura 3] sarebbe inutile eseguire questi test, oppure [Figura 4] ci sarebbe un errore nel testing.

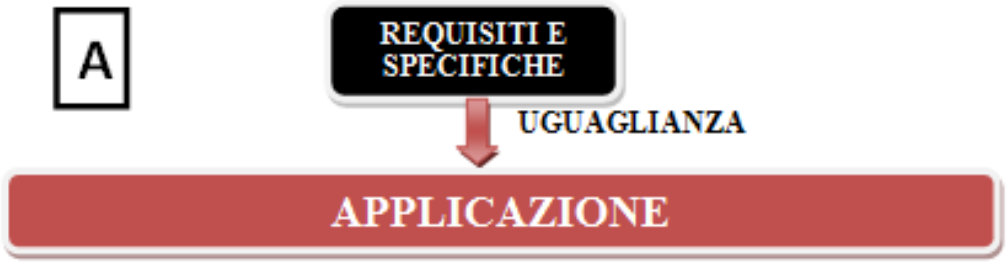

Figura 3 – *La documentazione non viene generata durante il processo di testing di un'applicazione che risponde perfettamente alle specifiche e ai requisiti.*

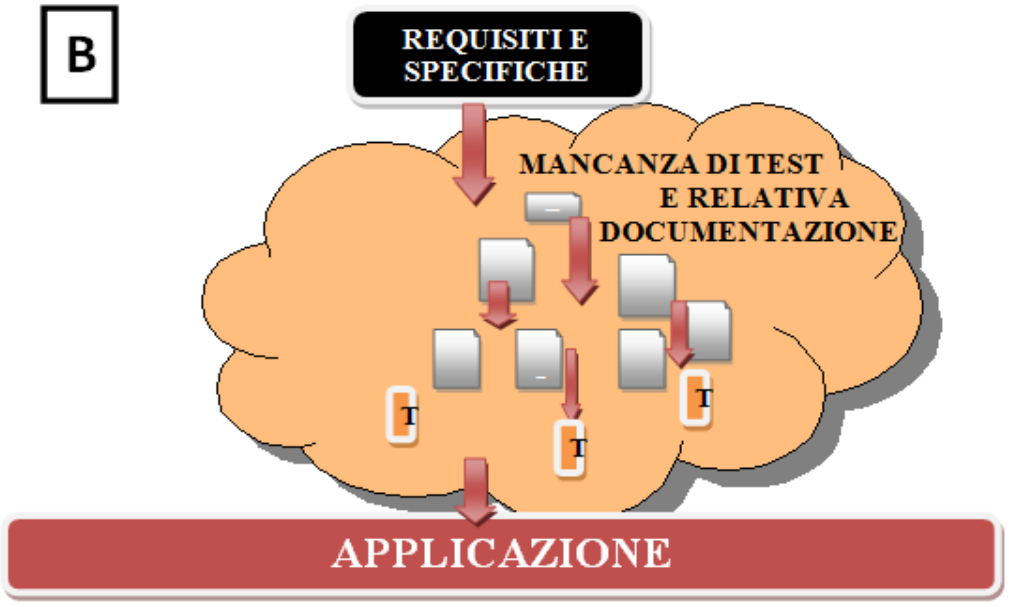

Figura 4 - *La documentazione dovrebbe essere generata, poiché il processo di testing presenta errori.*

Viceversa, un'attività di testing che prevede "n" test case, contenenti almeno una condizione negativa ciascuno, evidenzierà la presenza di "n" difetti nell'applicazione, che risulterà, quindi, difforme da "n" requisiti sui quali è stata definita [Figura 2].

La definizione del testing, come "pianificazione, progettazione, costruzione, manutenzione e realizzazione di test ed ambienti di test" data da Herzel non è mai stata così appropriata come in questo contesto.

Si può notare, infatti, come la presenza o meno dell'attività di testing possa essere indice di una certa misura di divario tra requisiti e realtà applicativa, esprimibile sottoforma di qualità.

Non solo, ma considerando che molte delle attività del processo di testing si basano sulla produzione e sullo studio di documenti, emerge che la documentazione è necessaria non solo per eseguire un buon testing, ma anche per garantire un alto standard qualitativo al prodotto software.

3.2.3 – Importanza della Documentazione di Test

Mentre il processo di testing stabilisce una certa misura per cui un'applicazione soddisfa i suoi requisiti, o meglio non li soddisfa, la documentazione generata, invece, descrive il modo con cui ciò avviene.

Per ogni requisito non soddisfatto, quindi ogni malfunzionamento verificatosi, tramite la documentazione è spesso possibile risalire al difetto applicativo che l'ha causato [17].

La presenza di documentazione offre vantaggi ancor più tangibili per i test di unità, dove lo scopo principale resta comunque il debugging, estremamente agevolato in fase di localizzazione degli errori nel codice e correzione degli stessi.

Ma non è tutto, poiché può non essere affatto intuitivo tradurre una mancanza o una difformità di un requisito con un difetto applicativo.

Può essere proprio una questione di uso di linguaggi differenti per la stesura e definizione dei requisiti e delle specifiche, rispetto alla sintassi e la semantica usati per definire gli stessi oggetti in un linguaggio di programmazione.

L'attività di testing sembra essere l'interprete più appropriato in questo contesto e la documentazione è il suo dizionario.

Inoltre, la documentazione di test si comporta come un mezzo di comunicazione sia all'interno del team incaricato al testing, che tra questo team e quello di sviluppo, dando una visione globale dell'attività, mettendo a fuoco i collegamenti con la realtà applicativa e spiegandone il significato.

Nonostante queste riflessioni facciano emergere l'importanza della documentazione di test e il ruolo fondamentale da essa giocato, spesso tali documenti risultano imprecisi, scorretti o addirittura mancanti.

Questo si spiega con il grande sforzo in termini di risorse, tempo e denaro necessario alla produzione e aggiornamento di tali documenti.

Considerando che l'attività di testing avviene poco prima del rilascio di un prodotto software, è soprattutto il tempo richiesto alla sua realizzazione ad essere visto come un ostacolo.

Proprio quando i tempi di consegna sono agli sgoccioli o addirittura sono sorpassati e magari il cliente desidera vedere subito in funzione il proprio applicativo, la documentazione di test viene abbandonata.

Inoltre, gli sviluppatori spesso considerano la generazione di tali documenti un compito poco gratificante.

Infatti, se generati manualmente, portano a compiere azioni meccaniche e ripetitive, privando lo sviluppatore di quella componente creativa tipica del suo ruolo.

3.2.4 – Mancanza di Documentazione nel Software Open Source

Nel caso in cui l'applicazione in fase di test sia un prodotto Open Source la produzione di documentazione completa e aggiornata risulta un compito ancora più arduo.

Lo sviluppo di tali prodotti coinvolge l'impegno di più programmatori dislocati in diversi luoghi, con uno stile e un approccio di lavoro differente e non appartenenti a realtà organizzate come invece accade per aziende, uffici o imprese.

Inoltre, la comunicazione tra gli sviluppatori spesso avviene mediante canali disorganizzati e non ufficiali, come mailing list e forum, che rendono difficile il coordinamento e l'uniformità di obbiettivi e metodologie utilizzate.

In questo scenario i dati di un Software Open Source risultano essere frammentari e non omogenei, rendendo ancora più difficile la creazione della relativa documentazione.

D'altra parte, però, è proprio la documentazione ad aiutare ad avere una visione globale del prodotto, descrivendolo dal punto di vista strutturale e operativo.

Infatti, uno dei primari motivi che frenano la scelta di utilizzare questo tipo di software, è che vengono spesso rilasciati senza manuali e documentazione delle specifiche.

Una conferma di quanto detto viene dall'analisi eseguita nell'ambito del progetto Qualipso [18], in cui sono stati osservati i portali di 32 importanti e conosciuti progetti Open Source.

Il 69% dei progetti possiedono manuali utente aggiornati, mentre nel 31% di essi questo tipo di documentazione risulta mancante o obsoleta.

Per quanto riguarda la documentazione tecnica i dati sono ancora più evidenti, infatti, mentre nel 49% dei progetti risulta adeguata, per il 51% di essi non è presente o non è aggiornata.

Nel contesto di testing queste difficoltà emergono con maggior forza, poiché questo tipo di prodotti sono in continua evoluzione e possono essere definiti dei requisiti non condivisi allo stesso modo da tutti gli sviluppatori.

Inoltre, le classiche metodologie di testing sono basate sulla separazione dei test dal codice applicativo, che comporta un'ulteriore disgregazione all'interno del codice di un prodotto Open Source.

Questo può portare alla mancanza di una linea comune su come deve avvenire il testing dell'applicativo.

La generazione della relativa documentazione di test assume un ruolo di fondamentale importanza, in quanto può aiutare ad avere una visione di insieme e a chiarire gli obbiettivi e le modalità del testing, facilitando il regolare svolgimento di tutte le fasi del processo.

Sempre in accordo con l'analisi condotta nel progetto Qualipso è emerso che solo JBoss [g], uno dei 32 progetti Open Source presi in esame, presenta una documentazione completa e aggiornata sulle attività di testing, sui piani e le metodologie di test, sui risultati ottenuti e una descrizione dei casi di test.

4.

CAPITOLO QUARTO

DOCUMENTAZIONE AUTOMATICA DEI CASI DI TEST

Come mostrato precedentemente, nonostante l'importanza ricoperta dalla documentazione di test, spesso essa viene sottovalutata o comunque si verificano diversi problemi che limitano la sua creazione ed il regolare aggiornamento.

Proprio per questo motivo, come trattato nella prima parte di questo capitolo, risultano interessanti i vantaggi offerti dalla generazione automatica della documentazione di test, che oltre a snellirne e facilitarne la creazione e l'aggiornamento, costituisce un supporto concreto all'intero processo di testing.

Successivamente si sono analizzati diversi strumenti che consentono la generazione automatica di documentazione, ed in particolare quello della Sun Microsystems, il tool Javadoc, per la creazione di documentazione tecnica di applicazioni Java.
Dopo aver messo in luce le sue caratteristiche di funzionamento ed estensibilità, e dopo aver considerato pregi e limiti, si è arrivati a definire un nuovo tool più adatto alla specifica documentazione di test.
Tale tool è stato chiamato T-Doc e nell'ultima parte del capitolo vengono presentati i motivi che hanno portato alla sua realizzazione, le sue caratteristiche, i dettagli dell'implementazione, la sua struttura interna ed i vantaggi offerti durante il suo utilizzo.

4.1 GENERAZIONE DELLA DOCUMENTAZIONE DI TEST

4.1.1 – Vantaggi della Generazione Automatica

La generazione automatica della documentazione può avvenire mediante l'uso di appositi strumenti software.
La maggior parte di essi prevede l'analisi del codice sorgente, opportunamente predisposto, per estrarne alcune informazioni rilevanti, al fine di ricavare un certo tipo di documentazione in modo automatico.
Nell'implementazione dei casi di test, così come in qualsiasi altro ambito della programmazione, è buona norma fornire commenti in linguaggio naturale all'interno del codice, per facilitarne la lettura, per spiegarne il funzionamento e per favorire la collaborazione tra diversi sviluppatori.
Se tali commenti vengono specificati con una sintassi standard, può allora essere costruito uno strumento in grado di generare la relativa

documentazione del codice in modo automatico, sfruttando la codifica di tali informazioni.

Questo non significa che basterà avviare lo strumento per aver risolto i problemi di creazione e aggiornamento della documentazione, ma anzi, che bisognerà prestare maggior attenzione alla stesura del codice da documentare, rispettando lo standard definito e rendendo i commenti più chiari, completi e dettagliati possibile.

In questo contesto, il commento assume un valore doppio, poiché oltre a spiegare il codice, è anche responsabile della sua documentazione.

Lo sviluppatore di una certa porzione di codice è la persona più adatta a spiegarne le caratteristiche, e se ciò avviene nel momento in cui il codice viene scritto, la generazione automatica permetterà di creare e aggiornare la documentazione in tempo reale.

Contestualmente al testing, è evidente che i vantaggi possono risultare molteplici, soprattutto in termini di risparmio di risorse.

I tempi sarebbero ridotti al minimo, velocizzando di conseguenza l'intero processo di testing, che, come visto in precedenza, ha la necessità di essere documentato in modo approfondito.

Il tempo risparmiato ha un grande valore nella fase che precede il rilascio di un prodotto, sia per motivi legati alle esigenze di eventuali clienti, sia per ragioni commerciali derivanti dalle caratteristiche del mercato informatico, in cui si tende ad anticipare la concorrenza.

Inoltre il tempo è denaro, nel ciclo produttivo del software, così come nella vita di tutti i giorni.

E' facile pensare che una rapidità maggiore nella produzione della documentazione possa tradursi in uno snellimento dell'intero processo di testing, causando un risparmio anche di risorse umane, che potrebbero essere destinate ad un raffinamento ulteriore del codice e ad un innalzamento della qualità del prodotto software.

Gli sviluppatori potrebbero concentrarsi sul codice, non dovendosi occupare di compiti gravosi e poco gratificanti come quello della stesura di documentazione.

Il vantaggio di poter creare e aggiornare documenti di test rapidamente e con uno sforzo minore si adatta perfettamente a quelle che sono le esigenze del testing di un applicativo, che viene spesso reiterato più volte e subisce variazioni in corso d'opera.

Inoltre, operando direttamente sullo strumento adibito alla generazione, è possibile ottenere una documentazione più approfondita, completa e coerente, senza dover prestare attenzione a ogni singolo elemento di documenti complessi come quelli che riguardano il testing, evitando, o perlomeno riducendo il rischio, di errori e imprecisioni.

L'automatizzazione, tra l'altro, favorisce la riusabilità di gran parte del lavoro svolto, in quanto è concentrato sullo strumento di generazione.

In un contesto tradizionale, dove lo sforzo consiste nel produrre documenti ad hoc per ogni singolo applicativo, ciò è chiaramente impossibile.

Inoltre, se consideriamo la generazione automatica di documentazione di test, in un contesto di prodotti Software Open Source, i vantaggi presentati risultano addirittura essere amplificati.

In questo ambiente, dove spesso il codice appare frammentato e poco omogeneo, la possibilità di produrre e aggiornare la documentazione, scaricando l'onere di tale compito sui commenti dei singoli sviluppatori, risulta essere di fondamentale importanza.

Ciò non è dovuto solo al fatto che risulta difficile scrivere un documento riferito ad un codice di questo tipo, ma soprattutto perché permette di avere una certa uniformità nella stesura documentazione.

La standardizzazione dei commenti e l'uniformità della documentazione possono anche contribuire alla creazione di un modello del processo di

testing condiviso tra i vari sviluppatori, supportando le varie fasi con esempi concreti della relativa documentazione inserita da altri e sviluppando delle regole comuni.

4.1.2 – Framework e Tool Esistenti

La generazione automatica della documentazione di test può avvenire mediante appositi tool creati a tale scopo, oppure attraverso un testing framework, ovvero una struttura di supporto su cui possono essere organizzati, progettati e valutati i test.

I tool di questo tipo, ad oggi maggiormente diffusi, sono Javadoc [h] e Doxygen [i].

Il primo è stato sviluppato dalla Sun Microsystems nel 1990, durante lo sviluppo del linguaggio Java e delle sue librerie.

Il gran numero di sorgenti spinse gli sviluppatori a creare questo strumento per la generazione automatica di documentazione tecnica in formato HTML (HyperText Markup Language), formato pratico e veloce per la consultazione.

Il tool venne poi distribuito insieme al kit di sviluppo Java, per la documentazione di progetti scritti in questo linguaggio.

Con il susseguirsi delle versioni, il tool divenne sempre più ricco e completo, introducendo, ad esempio, la possibilità di definire delle estensioni per personalizzare il contenuto e il formato di output della documentazione.

Doxygen, invece, fu sviluppato a partire dal 1997 da Dimitri van Heesch come progetto Open Source ed è un sistema multipiattaforma per la generazione di documentazione tecnica di un qualsiasi prodotto software.

Questo tool può quindi essere utilizzato su svariati sistemi operativi ed è in grado di operare con codice sorgente scritto in diversi linguaggi: C++, C, Java, Objective C, Python, IDL, PHP e C#.

Sia Doxygen che Javadoc sono stati creati con l'intento di analizzare il codice sorgente ed estrarre informazioni dalla dichiarazione di strutture dati e da commenti scritti con una particolare sintassi, usate per la generazione di API specifications (Application Programming Interface).
Quindi, all'interno del processo di testing, entrambi i tool sembrano essere adeguati al supporto della fase di implementazione, consentendo la generazione della documentazione tecnica relativa ai test case realizzati.
Tuttavia, l'adozione di questi strumenti, non permetterebbe comunque di ottenere in modo automatico la documentazione delle altre fasi del processo.
L'elevato grado di estensibilità del tool Javadoc, però, lo rende particolarmente interessante in questo contesto e verrà pertanto approfondito più avanti.
Anche diversi framework di testing Open Source forniscono supporto ad una o più specifiche fasi di tale processo, ma nessuno di essi rende disponibile la gestione, la creazione e l'aggiornamento di tutta la relativa documentazione.
Ad esempio, Testopia [j] fornisce un interessante funzionalità in grado di estendere Bugzilla, un tool dedicato al debugging, associando test case, relativi log di esecuzione e summary report dei difetti riscontrati.
Fitness [k], invece, è un altro tool che permette di semplificare la gestione di documenti, risultati di test e la creazione collaborativa di test di accettazione, ma oltre a evidenziare dei limiti per alcune tipologie di testing, non permette la generazione automatica di tutta la documentazione.
Anche framework più complessi, come TPTP [l] o Salome-TMF [m], pur supportando l'intero processo di testing, non prevedono la generazione automatica di tutti i documenti.

4.2 JAVADOC TOOL

4.2.1 – Generazione Automatica di Documentazione Tecnica

Lo scopo del tool Javadoc è quello di rendere possibile la generazione automatica di documentazione tecnica, detta anche API specification (Application Programming Interface), che rappresenta mediante un astrazione la struttura interna di un applicativo e le procedure a disposizione dello sviluppatore.

Per fare questo, viene effettuato un parsing dei file sorgenti, ovvero un analisi di tale codice, tramite il compilatore Java.

L'intento è quello di ottenere un certo numero di informazioni derivanti dalla dichiarazione di strutture di programmazione e da commenti espressi con una particolare sintassi, che verrà mostrata più avanti.

La Javadoc può essere eseguita su interi package, su singoli file o su entrambi.

Oltre al codice sorgente possono essere specificati altri file da utilizzare durante la generazione.

Il file html di overview, ad esempio, che fornisce informazioni generali sull'intero applicativo, viene collocato dalla Javadoc nell'apposita pagina, se viene specificata la sua posizione tramite un particolare comando.

Inoltre, la Javadoc rileva automaticamente, senza nessun comando, se sia presente o meno la panoramica di ogni singolo package, cercando nella relativa cartella un file nominato "package.html".

Infine, è possibile specificare, sempre all'interno della cartella del package, una sottocartella che deve avere il nome "doc-files", contenente altri file che possono essere importati nella documentazione, come immagini, esempi di codice o altri file html.

Sfruttando le informazioni nel codice sorgente e i file passati in input, il motore Javadoc standard consente di ottenere un insieme di pagine html, che descrivono overview, package overview, classi, interfacce, costruttori, metodi e campi.

Inoltre, vengono prodotte alcune pagine per i riferimenti incrociati tra diversi oggetti, come ad esempio: lista di nomi deprecati, uno schema delle dipendenze per ogni package e gerarchie delle classi sia per l'intero applicativo che per ciascun package.

Infine, sono incluse nella documentazione anche alcune pagine di supporto alla visualizzazione, come la pagina di indice, contenente i frame visualizzati, il foglio di stile CSS (Cascading Style Sheet), le cartelle "doc-files" specificate prima dell'esecuzione e una pagina contenente la guida alla navigazione del documento prodotto.

E' utile considerare che, durante l'esecuzione, non possono essere modificati o incorporati direttamente risultati di generazioni precedenti, tuttavia possono essere forniti dei link a tali risorse.

Nel contesto di un processo di documentazione tradizionale, le numerose fasi da intraprendere per ottenere un output simile a quello prodotto dalla Javadoc, oltre a risultare eccessive in termini di risorse necessarie alla loro attuazione, a volte sono addirittura irrealizzabili da agenti umani, che hanno un limite nella gestione della complessità.

Il processo di generazione della Javadoc, invece, consiste in 3 semplici fasi: la spiegazione del codice sorgente, da eseguire manualmente, la predisposizione del tool, che consiste nella preparazione manuale di eventuali file da associare al documento e la specifica di alcuni particolari comandi e opzioni che verranno applicati automaticamente, e la generazione vera e propria, che avviene in modo completamente automatico.

La spiegazione dei sorgenti avviene mediante i commenti in linguaggio naturale che lo sviluppatore deve preoccuparsi di inserire nel codice.
E' buona norma inserire sempre questo tipo di commenti per rendere il codice più chiaro e leggibile, perciò non viene chiesto allo sviluppatore di eseguire un compito aggiuntivo, ma solo di seguire particolari regole sintattiche.
Un commento Javadoc viene sempre espresso con un blocco di testo inserito tra i caratteri "/**" e "*/".
All'interno del blocco ogni riga è preceduta dal carattere "*".
Gli asterischi su ogni riga sono inseriti per differenziare un commento Javadoc, da un commento tradizionale [Figura 5], i quali rimangono quindi utilizzabili, ma non verranno inseriti nella documentazione.

```
1 /**
2  *Questo è un commento Javadoc che verrà inserito nella documentazione
3  */
4
5 /*
6   Questo è un commento standard che rimarrà solo nel codice Java
7 */
```

Figura 5 - *Commento per Javadoc e commento standard in Java.*

I commenti Javadoc possono contenere codice html e vengono posizionati nella riga che precede una qualsiasi dichiarazione di classe, interfaccia, metodo o campo, e verranno associati a tale elemento.

```
1 /**
2  *Questo è una breve spiegazione del metodo <b>testClass1()</b>
3  *
4  * @param index spiegazione del parametro(esempio di tag Javadoc)
5  */
6 public void testClass1(int index) {
7   ...
8 }
```

Figura 6 - *Esempio di commento Javadoc applicato ad un metodo.*

Come mostrato in [Figura 6], la struttura di un commento Javadoc prevede come primo elemento una breve frase riassuntiva, contenete una descrizione concisa ma completa dell'entità dichiarata.
Successivamente possono comparire uno o più tag preceduti dal simbolo "@", che verranno parsati durante la generazione della Javadoc.
I principali tag che possono essere inseriti in un commento sono:

TAG	DESCRIZIONE
@author	Viene seguito da una stringa che identifica l'autore dell'entità documentata
@deprecated	Identifica un'entità destinata a scomparire nelle future versioni del software ma ancora in uso, viene seguito da una descrizione testuale che solitamente indica l'entità con cui verrà rimpiazzata
@exception **@throws**	Indicano un'entità che può sollevare eccezioni, sono sinonimi e sono seguiti dal nome della classe, ovvero il nome dell'eccezione che può essere generata, e da una descrizione testuale.
@param	Indica un parametro passato nel costruttore dell'entità documentata, è seguito dal nome del parametro e da una descrizione testuale
@return	Specifica cosa può essere restituito come risultato dopo l'esecuzione dell'entità associata, viene seguito da una descrizione testuale
@see	Indica un link o un riferimento in relazione con l'entità documentata, può essere espresso in diversi modi: seguito da semplice testo, seguito da link html, oppure, nel caso di riferimenti ad altre entità della documentazione, nella forma "package.classe#metodo testoVisualizzato"
@version	Specifica la versione dell'entità documentata, spesso indicata con la versione del software nella quale è stata introdotta, è seguita da un testo descrittivo

Oltre a questi tag è possibile inserirne altri due particolari, detti tag inline.
La particolarità sta nella posizione che possono assumere all'interno del commento e di una sintassi diversa da quelli visti in precedenza.
Tali tag, infatti, possono essere inseriti in tutte le posizioni del commento in cui può essere scritta una qualsiasi stringa di testo, nella forma "{@tagInline}".

I principali tag inline che possono essere usati sono:

TAG INLINE	DESCRIZIONE
{@link}	Equivalente inline del tag @see, ma esprimibile solo nella forma {@link package.classe#metodo testoVisualizzato }
{@docRoot}	Rappresenta il percorso relativo per raggiungere la cartella principale di creazione della Javadoc da qualsiasi pagina generata, utile ad esempio per stabilire link e riferimenti a documenti esterni.

Una volta commentati in modo adeguato tutti i sorgenti, rimangono da specificare le impostazioni per la generazione della Javadoc.

Tali impostazioni prendono il nome di "option" e possono specificare ad esempio: la presenza della pagina di overview, un CSS diverso da quello standard, frammenti di codice html da porre in cima o alla fine di ogni pagina creata, il titolo del documento, le cartelle o i file sorgenti, la cartella in cui la Javadoc verrà creata, l'inclusione o meno dei tag "@author" e "@version" nella documentazione, la visibilità o invisibilità di elementi pubblici, protetti o privati.

Infine, l'opzione sicuramente più interessante, che verrà approfondita in seguito, è quella che permette la specifica di una "Doclet" diversa da quella standard.

Le Doclet sono estensioni di JavaDoc che permettono di gestire a piacimento le varie fasi di generazione della documentazione.

Per concludere, la documentazione può essere creata semplicemente eseguendo da riga di comando il tool Javadoc seguito dalla specifica delle option.

Inoltre, ormai tutti i moderni IDE (Integrated Developement Environment), che supportano il linguaggio Java, hanno Javadoc integrato tra i propri strumenti, rendendo la generazione un compito ancora più semplice e immediato [Figura 7].

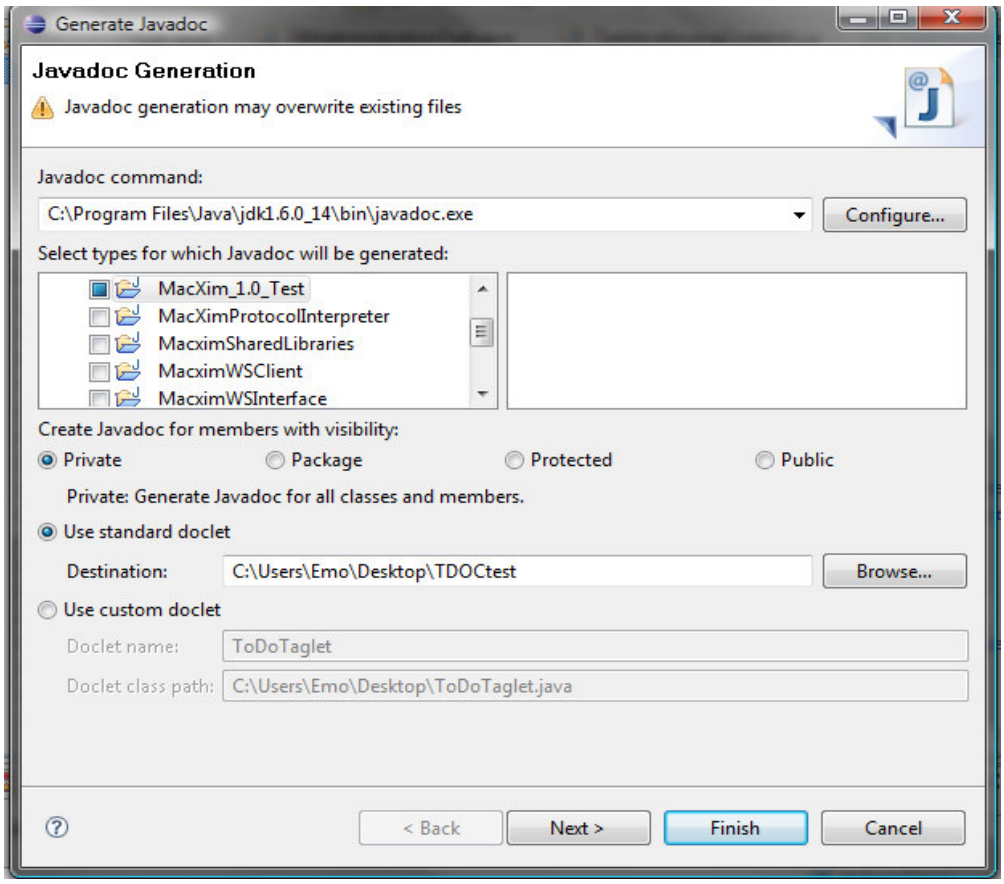

Figura 7 - *Generazione della Javadoc tramite Eclipse.*

4.2.2 – Estensibilità del Tool

Il punto di forza della Javadoc è sicuramente la possibilità di essere esteso e adattato per incrociare le diverse esigenze che si possono presentare durante la generazione di documentazione.

L'aspetto professionale, la struttura logica e l'organizzazione delle pagine html prodotte dalla Javadoc standard possono non essere sufficienti in alcuni contesti.

Le dinamiche di generazione della documentazione presentate in precedenza, in Javadoc sono implementate mediante "Taglet API" [o] e "Doclet API" [n].

La doclet è il motore vero e proprio per la generazione della documentazione, mentre i taglet sono usati dalla doclet per identificare i tag che verranno parsati nel codice e il relativo output che dovranno produrre nella documentazione.

L'alto grado di estensibilità della Javadoc ci permette di definire nuovi taglet e addirittura consente la scrittura di nuove doclet, per personalizzare l'output prodotto dalla Javadoc nel modo desiderato.

Per modificare il comportamento della Javadoc è quindi possibile agire in tre modi: inserire nuovi taglet nella doclet standard, modificare la doclet standard oppure creare una doclet propria.

L'aggiunta di nuovi taglet si effettua semplicemente scrivendo un piccolo programma che implementa l'interfaccia Taglet.

Specificando un nuovo taglet si permette alla doclet di riconoscere un nuovo tag nella versione classica (@tag), oppure di tipo inline ({@inlineTag}).

Tuttavia, bisogna tener presente che all'interno dei tag personalizzati non è possibile inserire i tag inline.

Tramite i taglet è possibile modificare e formattare il testo prodotto dalla Javadoc in corrispondenza dei tag ad essi associati, e anche altro, come ad esempio reindirizzare tale testo su un file.

Un taglet può anche sovrascrivere un tag standard già presente e per essere eseguito basta che venga specificato tramite due comandi di option: la localizzazione del suo file sorgente compilato ed il nome di tale file.

Per apportare modifiche più sostanziose, invece, è necessario scrivere un programma che faccia uso delle doclet API per specificare il contenuto e il formato dell'output della Javadoc.

Se non viene espressamente specificato nei comandi di option, la generazione della Javadoc produce file html mediante la doclet standard, che è composta da tre package: il primo si occupa di scrivere qualsiasi output html e di gestire i taglet (`com.sun.tools.doclets`), il secondo contiene un certo numero di classi per la creazione delle singole pagine html (`com.sun.tools.doclets.standard`), mentre il terzo contiene tutti i taglet disponibili (`com.sun.tools.doclets.standard.tags`).
Per cambiare il comportamento della doclet è possibile partire da questa struttura e modificarla, oppure riscriverla per intero.
Perché possa funzionare correttamente, è comunque necessario verificare l'importazione del package `com.sun.javadoc`, per poter usare le doclet API, e accertarsi che il punto di ingresso del programma sia una classe contenente il metodo `public static boolean start`, che prenda come parametro l'oggetto RooT-Doc.
Questo parametro prenderà le informazioni su qualsiasi option specificata in fase di esecuzione della Javadoc e sui file e package che devono essere documentati.
Come per un taglet, anche la doclet ha bisogno di essere compilata prima di essere eseguita e il suo utilizzo deve essere specificato mediante due appositi comandi di option, contenenti il percorso in cui si trova il codice e il nome del file.
La possibilità di definire taglet e doclet personalizzati permette, quindi, la gestione delle fasi di generazione della documentazione.
Questo, oltre a rendere la Javadoc un prodotto davvero competitivo nella creazione e aggiornamento della documentazione, ha permesso un rapido sviluppo del linguaggio Java, che ha reso disponibili fin dall'inizio le proprie API in modo chiaro e completo.

L'estensibilità della Javadoc ha permesso anche a case produttrici di software e sviluppatori Open Source di creare strumenti molto diversificati.
Ad oggi sono presenti doclet in grado di generare schemi UML (Unified Modeling Language), grafi di dipendenze fra classi e package, analizzatori di codice, e di produrre documentazione in diversi formati, come ad esempio PDF, Word, RTF e LaTex.

4.2.3 – Pregi e Limiti

Per quanto riguarda la documentazione del processo di testing, se da un lato possiamo considerare la bontà dello strumento Javadoc, nella sua versione standard, per la generazione automatica delle API dei test case, d'altra parte bisogna anche ammettere dei limiti.
Il risparmio di tempo e di risorse durante la creazione di questa documentazione e la conoscenza approfondita dei casi di test realizzati aiutano davvero ad avere una visione immediata, chiara e completa dello stato del testing dell'intero applicativo?
La Javadoc, in questo contesto, così come si presenta nella sua versione standard, se da un lato fa tirare un piccolo sospiro di sollievo al mondo dell'Open Source, dall'altro risolve ben poco.
Il modello Javadoc standard può, infatti, essere visto come una forma embrionale di coordinamento tra diversi sviluppatori e aggregazione del codice sorgente, può in qualche modo favorire una visione d'insieme e accrescere, anche se di poco, la definizione di standard operativi e di best-practices.
Questo tipo di documentazione, però, da sola non è sufficiente a rispondere alle esigenze che emergono dall'intero processo di testing, dove è utile conoscere e avere una visione di tutte le fasi che conducono dai requisiti definiti all'applicazione reale.

Il tool copre solo una fase di tale processo, la rappresenta fedelmente e nel dettaglio, ne considera diversi aspetti interessanti, ma non sembra affatto orientato a muoversi dinamicamente tra fasi differenti.

Proprio partendo dai vantaggi offerti nella fase di implementazione dei test, sarebbe utile replicare i benefici in tutte le fasi del processo ed estendere il modello su un'unica struttura organica, in grado di mantenere quei forti legami di dipendenza tra i documenti generati dal testing.

Alcune problematiche derivanti dall'applicazione del modello Javadoc, in un contesto di testing, potrebbero essere risolte tramite un'adeguata estensione ottenuta grazie a una nuova doclet.

Ad esempio, se un test restituisce un errore durante la sua esecuzione è ragionevole pensare che la soluzione potrebbe essere cercata con l'aiuto della documentazione tecnica, ma nel momento in cui il test evidenzia un fallimento, e quindi un probabile difetto nel software, come potrebbe rivelarsi utile un documento di questo tipo?

Se fossero documentati anche i possibili valori che porterebbero il test al successo o al fallimento, emergerebbe subito e in modo chiaro quale input ha provocato il risultato ottenuto.

Inoltre mentre nelle API non vi è distinzione tra un metodo ed un altro, nei test sarebbe utile capire la tipologia e la natura del test, poiché gli obbiettivi e le tecniche operative possono variare in modo sostanziale.

Ad esempio, mentre un blocco del sistema durante un test di unità deve evidenziare un errore o un fallimento, per un test di stress potrebbe essere una condizione normale e voluta, per verificare la reazione ed il tempo di ripresa dell'applicativo.

La documentazione prodotta tramite un estensione del modello Javadoc potrebbe tenere traccia di queste informazioni preziose in ambito di test e diversificare l'uso di alcuni strumenti già esistenti.

Ad esempio, la dipendenza e la correlazione tra metodi e componenti è già rappresentata nella documentazione prodotta dalla Javadoc, ma potrebbe essere molto utile anche durante i test di integrazione, se fosse opportunamente segnalato.

Altri limiti legati all'uso della Javadoc in un contesto di testing sono più difficilmente risolvibili mediante l'estensione di tale modello.

La prima considerazione su tutte, è che la documentazione del processo di testing avviene in tempi diversi in base alla fase in cui ci si trova.

Solitamente ogni documento è necessario alla fase successiva e implica la creazione di altri documenti, che a loro volta attiveranno altre fasi.

Un esempio su tutti sono i log ed i report ottenuti dopo l'esecuzione dei test, che possono essere ricavati dopo i documenti che hanno portato all'implementazione dei test.

In tale contesto pare difficile pensare che la Javadoc possa operare in tempi diversi da quello della sua esecuzione e soprattutto che possa incorporare automaticamente e riferirsi a documenti prodotti in precedenza.

Inoltre una caratteristica da non sottovalutare per la documentazione dei test è la presenza di uno storico, fondamentale per l'analisi e la valutazione dei progressi di un applicativo, che non possono essere mantenuti in modo automatico se non con l'ausilio di strumenti di memorizzazione persistente, come ad esempio database.

Un esempio di test che non possono fare a meno di questa modalità operativa sono quelli di regressione.

Ovviamente questo risultato può essere conseguito accostando una doclet particolare ad un sistema per la gestione e la memorizzazione dei dati, ma sembra andare ben oltre la semplice estensione del modello Javadoc.

Infine, la considerazione forse più importante in uno scenario di testing, è che la realtà documentata dei test può essere slegata da quella applicativa.

Se da un lato è vero che è possibile definire mediante Javadoc dei riferimenti ad una documentazione esterna, dall'altro bisogna pensare che non c'è modo di sapere se l'applicativo a cui ci si riferisce sia testato in ogni sua parte e per ogni funzionalità.
Concetti fondamentali, come quello di copertura del codice per i test di unità, non sembrano poter trovare spazio nel modello Javadoc.

4.3 T-DOC

4.3.1 – Motivazioni della Realizzazione

La soluzione ai limiti emersi da un uso del modello Javadoc in un contesto di testing, sembra portare alla creazione di un nuovo tool.

Tale tool è stato chiamato T-Doc [19] e vuole essere una risposta concreta ai problemi che possono emergere dalla generazione automatica della documentazione di testing.

La T-Doc sfrutta quelli che sono i vantaggi innegabili di uno strumento evoluto, ricco e completo come la Javadoc, adattandoli ad un contesto di testing e integrandoli in un nuovo strumento, composto da diversi moduli applicativi con funzionalità diversificate.

Proprio per questo motivo si vuole dare la possibilità alla T-Doc di muoversi con maggiore dinamicità tra i vari livelli di documentazione, facendo da supporto all'intero processo di testing e non solo alla documentazione delle API fine a se stessa.

Questo consentirebbe anche lo sviluppo di una visione globale del processo di testing, in modo condiviso e uniforme all'interno di un contesto Open Source, dove le problematiche legate alla documentazione si manifestano con maggior forza a causa della frammentazione del codice.

Per fare questo si è pensato ad un'estensione della doclet standard, per esempio, tramite il riconoscimento di nuovi tag per identificare i valori di successo e insuccesso di un test case, oppure per creare una sorta di legame con il piano di test, specificando la tipologia di ogni test.

Le dipendenze e le correlazioni tra metodi e componenti vogliono essere sfruttate per dare suggerimenti sui test di integrazione da eseguire.

L'intero tool vuole essere associato ad un supporto di memorizzazione delle informazioni che consenta di suggerire e valutare test di regressione.

Sempre tramite questo supporto sarà possibile operare sulla documentazione prodotta in istanti di tempo diversi, come ad esempio le informazioni ottenute dopo l'esecuzione dei test.

La T-Doc è costituita da tre layer principali, che si occupano di:

- la generazione automatica della documentazione dei casi di test,
- la generazione automatica di suggerimenti per le attività di testing di integrazione e regressione,
- e la generazione di report sui risultati di esecuzione dei test.

Inoltre, usando il tool T-Doc assieme all'approccio "Built In Test" (BIT), è possibile anche aggregare il codice applicativo con quello di test, facendo in modo che le due realtà possano rispecchiarsi l'una con l'altra.

Questa metodo, oltre a favorire l'aggregazione dei dati, può anche aiutare a standardizzare uno stile di programmazione comune per la definizione dei test, attraverso semplici regole.

Infatti, in questo approccio, per ogni componente vengono mantenuti insieme sia il codice dei test che quello applicativo.

Ogni classe potrà essere eseguita in due diverse modalità: quella normale, che renderà invisibile all'utente le funzionalità di test, e quella di manutenzione, in cui verranno abilitate le capacità di testing dei componenti.

Entrambe le modalità possono essere attivate mediante l'interazione con la rispettiva interfaccia applicativa e di testing.

4.3.2 – Documentazione Automatica dei Casi di Test

Il primo layer della T-Doc si occupa della generazione automatica di documentazione dei casi di test.

In questo contesto, la documentazione dell'attività di testing non vuole essere limitata ad una maggiore comprensione degli aspetti tecnici dei casi di test, come avviene per la Javadoc.

Lo scopo di questo layer, infatti, è la rappresentazione di un caso di test attraverso un unico documento che lo descriva nelle sue varie fasi di pianificazione, progettazione e implementazione.
Gli aspetti tecnici dell'implementazione sono quindi affiancati da dati inerenti la descrizione dei casi di test, quindi gli input forniti, gli output attesi, e le condizioni di superamento del test.
Altri dati documentati sono quelli tipicamente appartenenti al piano di test e alle specifiche dei test, come ad esempio la tipologia dell'attività di testing svolta (unità, regressione, integrazione, …) e gli aspetti qualitativi presi in esame con il test in oggetto (performance, funzionalità, sicurezza, …).
Nel contesto descritto, emerge un chiaro tentativo di sfruttare le informazioni provenienti da più documenti che si sarebbero ottenuti mediante un approccio classico alla documentazione di test, muovendosi in modo più dinamico tra le varie fasi del processo di testing [Figura 8].
L'aggregazione di tali informazioni di test, non solo consente una visione di insieme adatta a facilitare la comprensione dell'intera attività, ma favorisce anche il rafforzamento di quei legami che attraversano tutti i documenti generati dal testing.

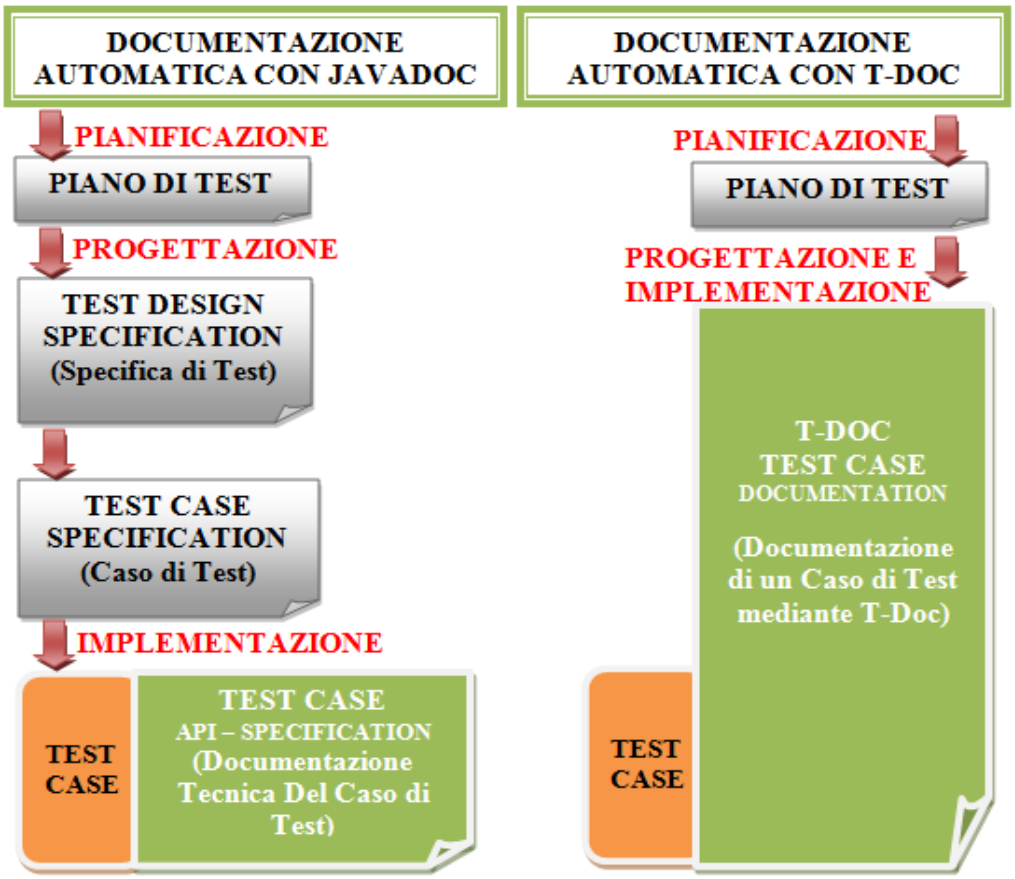

Figura 8 - *Esempio di documentazione automatica (rettangoli verdi) di un caso di test, ottenuta mediante Javadoc (a sinistra) e T-Doc (a destra).*

Ogni caso di test, documentato mediante T-Doc, può essere associato in modo chiaro e diretto al piano di test, grazie alle informazioni esplicite sulla sua tipologia, lo scopo e le caratteristiche di qualità che esso intende verificare.

Questo implica anche una forma di standardizzazione nella definizione dei test, caratteristica di fondamentale importanza nel testing di prodotti Open Source, carenti sotto questo aspetto.

La generazione automatica avviene in modo simile a quanto visto per la Javadoc, avvalendosi però di una doclet opportunamente modificata.

Le informazioni necessarie alla creazione della documentazione sono quindi contenute nei blocchi di commento del codice sorgente, posizionati subito prima dei casi di test che si intende documentare.

La struttura di un blocco di commento è esattamente quella prevista per il tool Javadoc, ma sono stati introdotti nuovi tag e alcuni già in uso assumono un significato differente per adattarsi al contesto in cui operano.

I tag significativi per il commento di un caso di test sono:

TAG T-DOC	DESCRIZIONE
@succeedIf	Riassume le condizioni sotto le quali il test ha successo, ad esempio gli input inseriti e i relativi risultati attesi.
@failIf	Riassume le condizioni sotto le quali il test fallisce, ad esempio gli input inseriti e i relativi risultati non attesi.
@scope	Specifica lo scopo del test, in particolare identifica la tipologia di testing al quale esso è associato, ad esempio: test di unità, di integrazione o di sistema.
@qualityAttribute	Specifica l'aspetto qualitativo che il test intende verificare, come ad esempio: performance, funzionalità o sicurezza.
@param	Indica un parametro passato in input al test, è seguito dal nome del parametro e da una descrizione testuale
@return	Specifica cosa può essere restituito come risultato dopo l'esecuzione del test, viene omesso nei test che restituiscono void
@author	Identifica il tester, la persona incaricata di eseguire il test in oggetto.
@see	Identifica la funzionalità testata, che viene specificata nella forma "package.classe#metodo testoVisualizzato"
@version	Specifica la versione del test documentato, non la versione del software testato, in quanto il verrà presumibilmente eseguito su più versioni dello stesso applicativo, si esprime nella forma "numeroVersione + dataCreazioneTest"

Sono quindi stati inseriti all'interno della doclet standard del tool Javadoc dei nuovi taglet per i tag @scope, @qualityAttribute, @failIf, @succeedIf, mediante la specifica di quattro file Java di configurazione.

All'interno di ciascun file [Figura 9], viene specificata una classe che implementa l'interfaccia Taglet e viene determinato in quali blocchi di commento deve essere riconosciuto il tag personalizzato, in particolare prima della dichiarazione di campi, costruttori, metodi, tipi, e nei file di overview o package overview.

Successivamente viene impostato che non si tratta di tag inline.

Infine, viene specificato come deve avvenire la rappresentazione del tag personalizzato nella documentazione HTML, sia nel caso in cui sia presente un singolo tag di quel tipo, sia nel caso in cui vi siano più tag dello stesso tipo.

```java
package org.uninsubria.tdoc;

import com.sun.tools.doclets.Taglet;
import com.sun.javadoc.*;
import java.util.Map;

/**
 * A sample Taglet representing @scope. This tag can be used in any kind of
 * {@link com.sun.javadoc.Doc}.  It is not an inline tag. The text is displayed
 * in yellow to remind the developer that is a test case.  For
 * example, "@scope unit" would be shown as:
 * <DL>
 * <DT>
 * <B>Test Scope:</B>
 * <DD><table cellpadding=2 cellspacing=0><tr><td bgcolor="yellow">unit
 * </td></tr></table></DD>
 * </DL>
 *
 * @author Jacopo Emoroso
 */
public class ScopeTaglet implements Taglet {

    private static final String NAME = "scope";
    private static final String HEADER = "Test Scope:";

    /**
     * Return the name of this custom tag.
     */
    public String getName() {
        return NAME;
    }

    /**
     * Will return true since <code>@scope</code> can be used in field documentation.
     */
    public boolean inField() {
        return true;
    }

    /**
     * Will return true since <code>@scope</code> can be used in constructor documentation.
     */
    public boolean inConstructor() {
        return true;
    }

    /**
     * Will return true since <code>@scope</code> can be used in method documentation.
     */
    public boolean inMethod() {
        return true;
    }

    /**
     * Will return true since <code>@scope</code> can be used in method documentation.
     */
    public boolean inOverview() {
        return true;
    }

```

```
61      /**
62       * Will return true since <code>@scope</code> can be used in package documentation.
63       */
64      public boolean inPackage() {
65          return true;
66      }
67
68      /**
69       * Will return true since <code>@scope</code> can be used in type documentation (classes or interfaces).
70       */
71      public boolean inType() {
72          return true;
73      }
74
75      /**
76       * Will return false since <code>@scope</code> is not an inline tag.
77       */
78
79      public boolean isInlineTag() {
80          return false;
81      }
82
83      /**
84       * Register this Taglet.
85       */
86      public static void register(Map tagletMap) {
87         ScopeTaglet tag = new ScopeTaglet();
88         Taglet t = (Taglet) tagletMap.get(tag.getName());
89         if (t != null) {
90             tagletMap.remove(tag.getName());
91         }
92         tagletMap.put(tag.getName(), tag);
93      }
94
95      /**
96       * The representation of this custom tag in source code,
97       * return its HTML string representation in the documentation.
98       */
99      public String toString(Tag tag) {
100         return "<DT><B>" + HEADER + "</B><DD>"
101                + "<table cellpadding=2 cellspacing=0><tr><td bgcolor=\"yellow\">"
102                + tag.text()
103                + "</td></tr></table></DD>\n";
104     }
105
106     /**
107      * An array of representations of this custom tag in source code,
108      * return its HTML string representation in the documentation.
109      */
110     public String toString(Tag[] tags) {
111         if (tags.length == 0) {
112             return null;
113         }
114         String result = "\n<DT><B>" + HEADER + "</B><DD>";
115         result += "<table cellpadding=2 cellspacing=0><tr><td bgcolor=\"yellow\">";
116         for (int i = 0; i < tags.length; i++) {
117             if (i > 0) {
118                 result += ", ";
119             }
120             result += tags[i].text();
121         }
122         return result + "</td></tr></table></DD>\n";
123     }
124 }
```

Figura 9 - *File contenente la classe che definisce il taglet relativo al tag @scope.*

Tali file di configurazione vengono associati alla doclet standard della Javadoc, tramite particolari comandi di option specificati prima della generazione [Figura 10].

I comandi di option di tipo "-taglet" indicano ciascun taglet aggiunto, mentre i comandi "-tagletpath" il percorso in cui si trova il relativo file di configurazione.

Infine, anche i tag @author e @version, che di default sono disabilitati nella doclet standard, devono essere attivati mediante altri comandi di option di tipo "-tag".

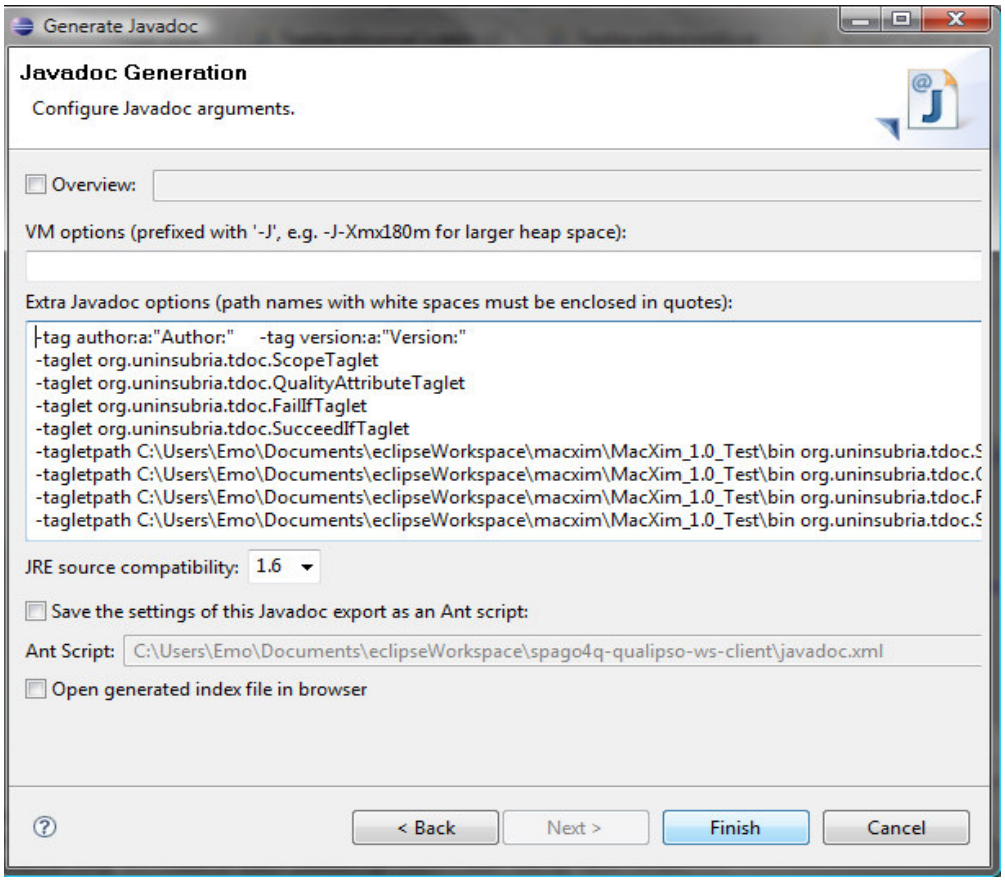

Figura 10 - *Specifica dei comandi di option per la generazione di documentazione secondo lo standard T-Doc in Eclipse.*

Figura 11 - *Codice sorgente di un metodo di test (A) e relativa generazione automatica della documentazione (B), secondo lo standard T-Doc.*

La generazione della documentazione mediante T-Doc [Figura 11] avviene nel momento in cui uno o più sviluppatori aggiungono o modificano classi con built-in test all'applicazione.

Ciascuna classe viene analizzata separatamente dal motore T-Doc [Figura 12], per scoprire e isolare i casi di test, che saranno sempre accompagnati dai loro commenti T-Doc.

Il componente "Test Suite Builder" si occupa di aggregare tutti i test built-in in un'unica test suite, mentre il componente "T-Doc TCs" si preoccupa

di effettuare il parsing di tutti i sorgenti dei test, per procedere con la generazione automatica della documentazione secondo lo standard T-Doc. La documentazione prodotta riguarderà, quindi, l'intera test suite e sarà sempre aggiornata e in linea con il codice applicativo e di test.
Infine, il motore T-Doc pubblicherà la documentazione su un repository centralizzato del progetto, in modo che essa possa essere fruibile da tutti e in tempo reale.

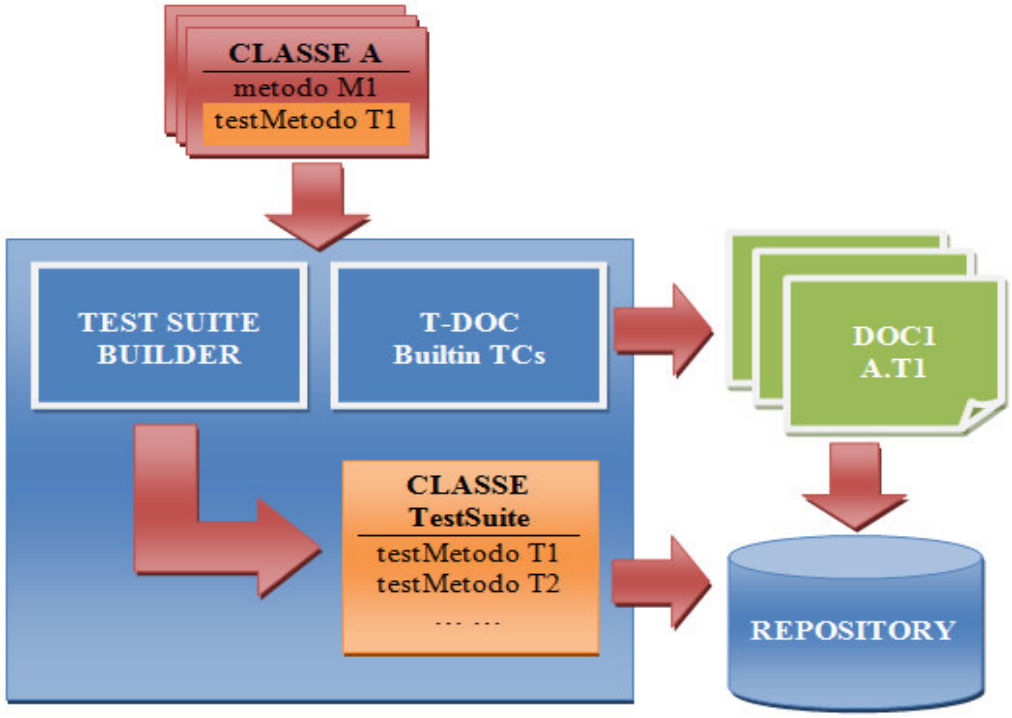

Figura 12 - *Architettura del primo layer T-Doc.*

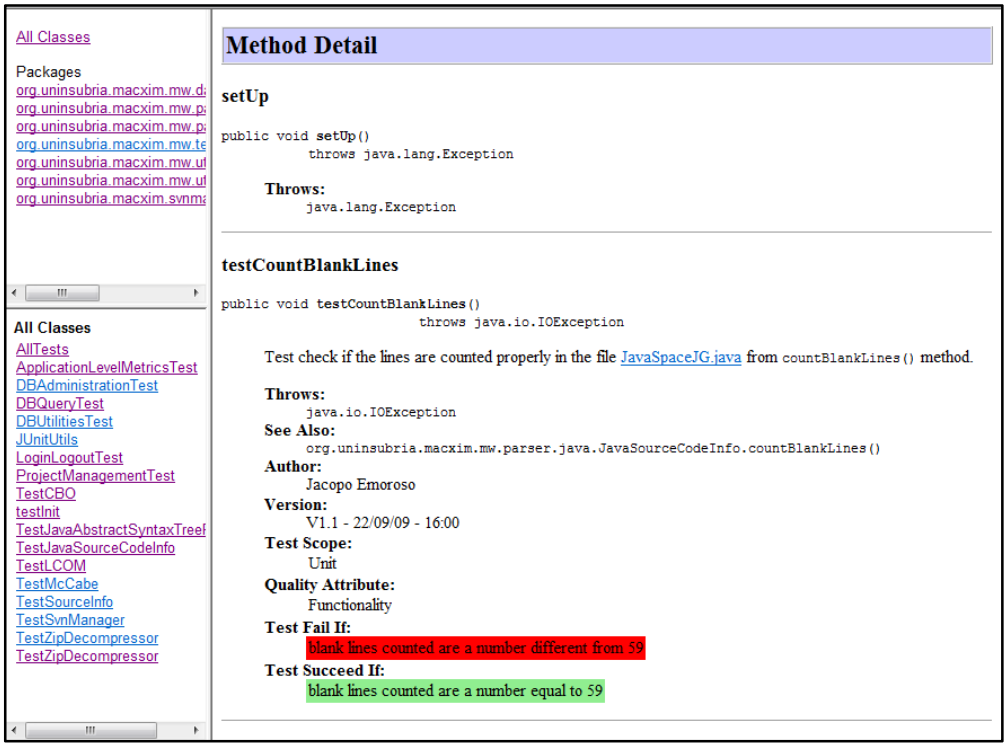

Figura 13 – *Esempio di documentazione HTML prodotta dal primo layer T-Doc.*

4.3.3 – Suggerimenti Automatici per Test di Integrazione e Regressione

Il secondo layer della T-Doc aiuta nel suggerimento e nella documentazione dei test di integrazione e regressione che gli sviluppatori dovrebbero generare durante l'aggiornamento e la manutenzione dell'applicativo.

Per fare questo la T-Doc trova automaticamente le modifiche avvenute nel codice sorgente e le dipendenze esistenti tra metodi e componenti, e sfrutta tali informazioni per mostrare graficamente allo sviluppatore i suggerimenti per i test di regressione ed integrazione.

Più precisamente la T-Doc sfrutta l'idea dei "change points" (CP) [20], ovvero i punti di cambiamento, che sono le parti del codice sorgente aggiunte o in cui si verificano delle modifiche.
La granularità con cui vengono considerati i CP è a livello di metodo, quindi in seguito ad un qualsiasi cambiamento al suo interno, tale metodo verrà identificato come CP.
Partendo da ogni singolo CP è possibile ricavare il relativo "call graph" (CG) ad esso associato, che consentirà di determinare le chiamate in relazione con il metodo considerato.
Un CG è, infatti, un grafo orientato che rappresenta le differenti procedure di un programma con dei nodi e le chiamate mediante una freccia uscente dalla procedura chiamante ed entrante nella procedura chiamata.
Considerando che, ogni qual volta un metodo verrà aggiunto o modificato all'applicazione, esso verrà riconosciuto come CP, il suo CG identificherà i possibili metodi influenzati da tale cambiamento.
In questo contesto, è facile immaginare come CP e CG possano essere il punto di partenza per la creazione ed il suggerimento di nuovi scenari di testing di integrazione e regressione.
I test di Integrazione, ad esempio, hanno il compito di mostrare come il livello qualitativo del software si sia mantenuto anche dopo l'introduzione o la modifica di un nuovo componente.
Se il nuovo componente ha superato i test di unità, non è escluso che possano emergere difetti nel software in relazione alle dipendenze che intercorrono tra esso e i componenti già presenti.
Il modulo "T-Doc Integration" si occupa proprio di trovare tali dipendenze, prendendo in input la documentazione generata dal modulo "T-Doc TCs", per identificare il CP, e generando il CG associato al caso di test documentato.

Per limitare le dimensioni del grafo, e un probabile carico computazionale eccessivo, si è scelto di considerare le dipendenze tra metodi fino al terzo livello.

Ad esempio, in seguito ad un cambiamento al metodo M1(), e quindi al relativo test T1(), il modulo "T-Doc Integration" produrrà automaticamente un documento che suggerirà di eseguire test di integrazione su tutti i metodi che interagiscono direttamente o indirettamente con M1 [Figura 14].

Tali metodi vengono ricavati grazie all'integrazione del Call Graph tool [q] nella T-Doc, che permette di creare automaticamente un CG a partire da un CP.

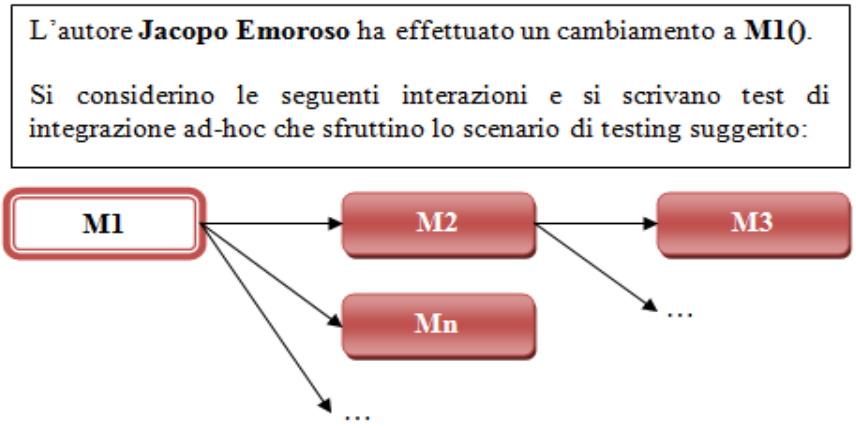

Figura 14 - *Scenario di testing di integrazione suggerito per il test T1().*

Per quanto riguarda il testing di regressione, invece, il suggerimento che si vuole ricavare automaticamente permette di determinare quali siano effettivamente i test da rieseguire per verificare la differente qualità dell'applicativo prima e dopo il cambiamento.

Per applicativi di medie e grandi dimensioni l'esecuzione dell'intera test suite può durare un tempo quantificabile in diverse ore, ma è sempre utile rieseguire tutti i test?

Specialmente in ambienti Open Source risulta difficile stabilire quando vadano rieseguiti i test e chi si deve far carico di tale compito.

Il modulo "T-Doc Regression" si occupa di risolvere tali problemi, segnalando, nel momento in cui si verifica un cambiamento, quali siano i test che effettivamente potrebbero avere risultati differenti rispetto all'ultima esecuzione.

Tale modulo prende in input il CP precedentemente identificato e un CG relativo a ogni test case già presente nella test suite, ricavato tramite il Call Graph tool.

E' importante notare che il CG ricavato per il CP ha come radice il metodo in cui è avvenuto il cambiamento (quindi esattamente il CP), mentre i grafi ricavati per i test case abbiano come radice il metodo di test a cui si riferiscono.

Il modulo "T-Doc Regression" ha il compito di cercare nell'insieme dei grafi relativi ai test case, tutti quelli che contengono il CP.

Il sottoinsieme dei grafi trovati rappresenta esattamente i test case che dovranno essere inseriti nella test suite di regressione.

La test suite di regressione contiene i test che possono essere influenzati dal cambiamento introdotto nell'applicativo e pertanto sono gli unici a dover essere eseguiti nuovamente per verificare eventuali differenze qualitative.

Gli output prodotti da questo modulo sono sia una documentazione testuale, in cui vengono riportati i test da eseguire, che la vera e propria test suite di regressione, che unitamente al documento prodotto dal modulo "T-Doc Integration" verranno pubblicati sul repository centrale.

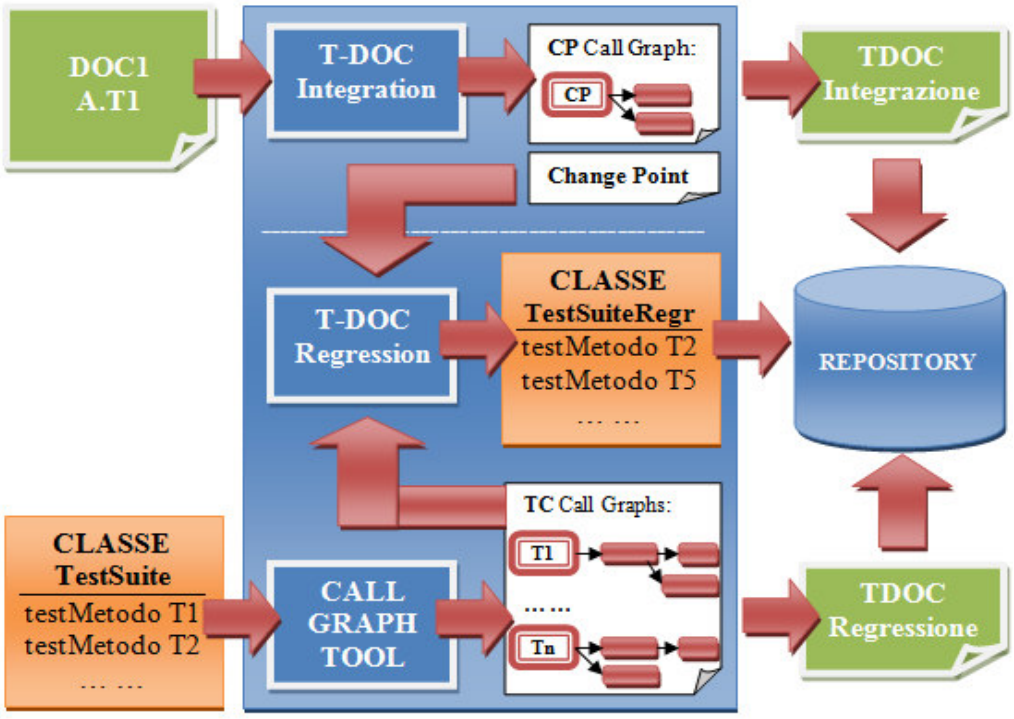

Figura 15 - *Architettura del secondo layer T-Doc.*

4.3.4 – Documentazione Automatica dei Report di Esecuzione dei Test

Il terzo layer della T-Doc si occupa di raccogliere i risultati di esecuzione dei casi di test e soprattutto di rendere disponibili in modo omogeneo i dati provenienti dagli altri layer.

I risultati di esecuzione, infatti, perdono di significato se non vengono affiancati dalle informazioni precedentemente elaborate dalla T-Doc.

Non è un caso, quindi, che questo layer in primo luogo abbia lo scopo di raccogliere la documentazione dei singoli casi di test, generando un unico file in grado di descrivere il comportamento dell'intera test suite.

Anche i suggerimenti per i test di integrazione e regressione vengono accorpati in un unico documento.

Inoltre, le differenti test suite archiviate nel repository possono essere agevolmente eseguite ripetutamente con differenti configurazioni hardware e software [Figura 16].

La disponibilità di test suite completa e test suite di regressione, consente e favorisce un immediato confronto dei risultati ottenuti durante la loro esecuzione.

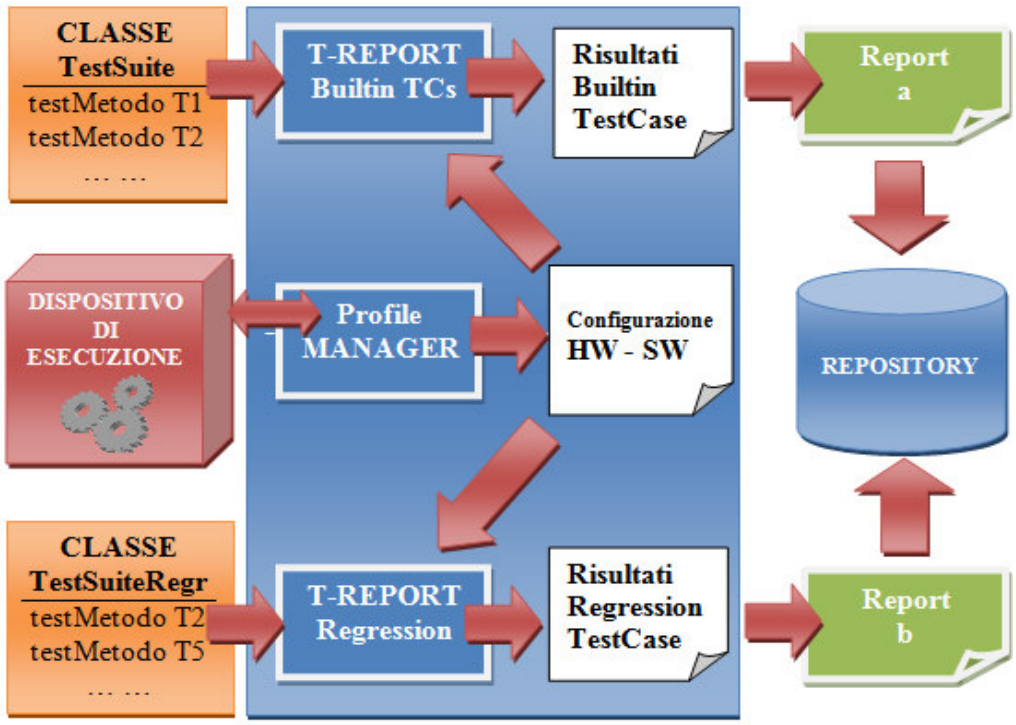

Figura 16 - *Architettura del terzo layer T-Doc.*

Il modulo "Profile Manager" consente di aggiungere informazioni preziose ai risultati prodotti dai singoli test case in ambienti hardware e software differenti.

Tali informazioni vengono filtrate e accorpate agli esiti dei test grazie ai rispettivi moduli "T-Report Builtin TCs", per la test suite builtin, e "T-Report Regression", per la test suite di regressione.

Entrambi i documenti prodotti da questi due moduli, contenenti configurazioni e rispettivi esiti, verranno poi pubblicati sul repository centrale.

Infine, questo layer fornisce anche un meccanismo di ricerca all'interno di tali documenti e nel repository centrale.

La ricerca all'interno di un sistema con una precisa struttura, come quello creato dalla T-Doc, porta dei vantaggi non indifferenti.

La possibilità di sfruttare i tag definiti nel primo layer consente di muoversi agevolmente attraverso più livelli del processo di testing, come se ci si trovasse davanti alla documentazione completa di pianificazione, progettazione, implementazione ed esecuzione.

L'aggregazione di documentazione proveniente da diverse fasi del processo di testing, oltre a favorire una visione di insieme efficace e ben strutturata, agevola le operazioni di check dei test, la valutazione e il debugging.

Il check dei test prevede il controllo della test suite, quindi il grado con cui il testing verifica le caratteristiche di qualità desiderate.

Questa attività si basa prevalentemente sulla consultazione incrociata di documenti di progettazione, pianificazione ed esecuzione.

La valutazione dei risultati ottenuti, invece, si accontenta dei report di esecuzione finché i test hanno successo, ma quando insorgono errori o fallimenti è utile consultare documentazione tecnica, di progettazione e di pianificazione, per poter risalire alle cause.

Infine, il debugging non può fare a meno di avere una visione completa del processo di testing, poiché la ricostruzione della causa di un fallimento di un test può essere trovata solo comprendendo a fondo le dinamiche che avvengono a run-time.

Il supporto che fornisce l'uso della T-Doc può essere quindi considerato fondamentale sull'intero processo di testing, in quanto gli obbiettivi di

questa attività possono essere raggiunti con facilità solamente incrociando informazioni provenienti da fasi distinte nel tempo e nelle dinamiche operative.

5.

CAPITOLO QUINTO

CONTESTO APPLICATIVO

Per fornire una prova empirica della validità del metodo T-Doc, si è deciso di applicare l' implementazione del primo layer del tool ad un progetto Open Source reale.

L'implementazione della T-Doc al progetto Macxim [r] consentirà di mostrare l'effettivo funzionamento, i reali benefici e il livello di automazione fornito dal tool.

In questo capitolo viene descritto lo scopo e il funzionamento di Macxim, per avere una prima idea dei requisiti applicativi.

Successivamente è stato definito un piano di test, specificando le attività da intraprendere, una prima identificazione dei test da implementare, il

testing framework che si è scelto di utilizzare e un progetto che costituirà l'input per i nostri test.
E' stata descritta la test suite implementata per Macxim, ed in particolare i test di unità, quelli di sistema e un insieme di test di accettazione interni al progetto.
Si è quindi arrivati a generare la documentazione dei casi di test mediante il tool T-Doc, potendo osservare il primo layer in esecuzione.
Tale generazione ha prodotto risultati concreti, i quali sono stati analizzati e ci hanno permesso di confermare alcuni vantaggi nell'utilizzo di tale modello.

5.1 IL PROGETTO MACXIM

5.1.1 – La Misurazione del Software

Macxim nasce per merito del progetto "QualiPSo", sponsorizzato dalla Comunità Europea, ed il progetto "La qualità nello sviluppo del software", sponsorizzato dall'Università degli Studi dell'Insubria.
Si tratta di una piattaforma Open Source gratuita, per l'analisi statica del codice sorgente e per la misurazione della qualità del software in linguaggio Java.
L'analisi di un progetto tramite Macxim inizia con l'estrazione di una rappresentazione del codice sorgente in un "Abstract Syntax Tree" (AST).
L'AST viene codificato in formato XML (eXtensible Markup Language), ed i file vengono salvati sul database XML nativo "eXist" [s].
Il database può essere interrogato mediante query XPath, o con query complesse implementate direttamente in Java.
Le query restituite sono anch'esse in formato XML e possono essere subito scartate oppure immagazzinate a database.

Le misurazioni possono essere effettuate su un singolo progetto, su più progetti contemporaneamente, oppure su più versioni di uno stesso progetto.

La caratteristica che distingue Macxim, da altri progetti per la misurazione della qualità del software, è data dall'elevato grado di estensibilità del tool, che permette la semplice definizione di nuove metriche e l'introduzione di nuovi parser per altri linguaggi di programmazione.

Le metriche già implementate per Macxim sono in parte native ed in parte provengono da tool esterni.

Ad oggi i tool esterni che Macxim può gestire sono PMD [t], Checkstyle [u] e FindBugs [v], i quali ricavano misure di complessità, correttezza e caratteristiche sullo stile di scrittura di codice sorgente.

Le metriche native correntemente implementate in Macxim sono:

METRICHE NATIVE DI MACXIM	
Num Package	Num. Classi
Num Classi con Metodi Definiti	Num Classi con Attributi Definiti
Num Classi Astratte	Num Interfacce
Num Interfacce non Implementate	Interfacce Implementate per Classe
Num Metodi	Num Metodi Pubblici
Num Metodi Privati	Num Metodi Protetti
Num Metodi per Classe	Num Metodi per Interfaccia
Num Parametri per Metodo	Num Attributi per Classe
Num Attributi Pubblici per Classe	Num Linee di Codice Effettive (eLOC)
Num eLOC per Classe	Num Linee di Commento
Num Linee di Commento Inline	Num Linee di Commento per Classe
Linee di Commento Inline per Classe	Linee di Commento per Interfaccia
LCOM (per Classe)	CBO (per Classe)
RFC (per Classe)	McCabe (Complessità per Metodo)

Le metriche possono essere calcolate, dove possibile, a quattro diversi livelli di granularità: per tutta l'applicazione, per un singolo package, per una singola classe o per un metodo.
Inoltre, per ogni metrica Macxim fornisce dei dati statistici come: valore massimo, valore minimo, deviazione standard, media e mediana.
L'utente finale può interagire direttamente con Macxim, attraverso un apposita interfaccia grafica rilasciata insieme all'applicativo, oppure tramite un servizio web service, che si appoggia sulla piattaforma Open Source Spago4Q [w].

5.2 IL PIANO DI TEST

5.2.1 – Attività di Testing Previste

La stesura del piano di test ha avuto come primo obiettivo quello di determinare le attività di testing da eseguire per il progetto Macxim.
In questo contesto, si è tenuto presente che il progetto Macxim è realizzato da un team di sviluppo composto da diverse persone.
Pertanto si è ritenuto utile che l'intera attività di testing fosse basata sulla misurazione di un progetto di test di piccole/medie dimensioni, condiviso da tutti e disponibile su un repository centrale.
Si è poi deciso di mantenere nel piano solo informazioni generali sulle varie attività di testing, come ad esempio la tipologia, le modalità di esecuzione, gli ambienti di test, framework e altri software da utilizzare.
Date le dimensioni di Macxim, nell'ordine di oltre 150 classi e una trentina di package, anche i test da realizzare sono indicati con caratteri generali, senza entrare nei dettagli che emergeranno in fase di progettazione.
La cosa non è banale, in quanto tali informazioni, normalmente presenti nei documenti di specifica dei test e dei singoli casi di test, vengono solitamente incorporate e ritenute parte del piano di test.
I dati, che verranno intenzionalmente omessi, vogliono essere generati automaticamente dalla T-Doc, che porterà inevitabilmente ad una fusione della fase di progettazione ed implementazione dei test.
Ciò non significa che verrà tralasciata la fase di progettazione per procedere immediatamente con l'implementazione, ma che essa verrà formalizzata direttamente all'interno del codice sorgente, mediante la stesura dei commenti T-Doc.
La prima attività di testing prevista all'interno dello schedule del piano di test è quella dei test di unità.

E' importante notare come non ci sia nessuna indicazione sulla priorità o criticità dei test, ne in questa attività ne nelle successive, in quanto l'imminente scadenza per il rilascio della prima versione dell'applicativo ha imposto una implementazione dei soli test di livello "critical", quindi strettamente necessari.

Per quanto concerne i test di unità si è voluto dare una copertura totale al codice sorgente, imponendo la presenza di almeno un test per ogni metodo realizzato, ed almeno un caso di test per input validi e uno per input non validi.

Ad ogni sviluppatore è stato imposto di eseguire il test di unità relativo al proprio codice applicativo realizzato, suggerendo di operare in parallelo sia sui test che sul codice sorgente.

I test realizzati vogliono essere conformi a quanto definito nello standard dei commenti T-Doc, sia nelle loro caratteristiche di progettazione, che sotto il profilo tecnico di implementazione.

Qualora questi test richiedessero l'analisi di codice sorgente, è stato imposto l'uso del progetto di test condiviso, di cui sono state rese note le specifiche caratteristiche misurate.

Inoltre, si è deciso di implementare tutti i test di unità con l'aiuto del framework di testing JUnit.

Una volta terminati i test di unità, lo schedule prevede lo svolgimento del testing di integrazione con modalità analoghe a quelle definite precedentemente, in base a quanto emerso dalla documentazione T-Doc generata.

Per quanto riguarda i test di sistema, invece, si sono determinate delle caratteristiche da testare a partire dai requisiti specifici dell'applicazione.

Sono stati favoriti test funzionali e di sicurezza, dando meno spazio a test prestazionali e di stress.

Il motivo di questa scelta è che i requisiti della prima versione di Macxim prevedono caratteristiche funzionali, mentre nelle successive versioni vengono introdotti concetti come miglioramento delle performance ed efficienza.

Proprio per questo motivo abbiamo chiamato l'insieme di questi test "Acceptance Test", in quanto non comprendono tutti i test di sistema, ma possono essere visti come una sorta di test di accettazione interna al gruppo di sviluppo, per validare la prima versione di Macxim.

A partire dai requisiti applicativi, nel piano sono stati tracciati tre principali gruppi di questo tipo di test: quelli relativi alla sicurezza, quelli relativi alle operazioni di gestione dei progetti e quelli che riguardano l'analisi dei progetti.

I test di sicurezza si occupano di verificare il corretto funzionamento di Macxim durante le operazioni di login e logout.

I test per la gestione dei progetti da analizzare testano le funzionalità di caricamento e cancellazione di un progetto, la richiesta dell'elenco dei progetti caricati e l'ottenimento dei dati generali relativi ad un progetto.

I test per l'analisi di progetti controllano che le misurazioni del codice sorgente avvengano correttamente per ogni livello di granularità selezionato (applicazione, package, classe e metodo).

Di seguito vengono riportati i dati relativi ai test di accettazione, così come sono stati definiti nel piano di test:

****MACXIM 1.0 – ACCEPTANCE TEST****			
TEST NAME	DESCRIPTION	EXPECTED RESULTS	
APPLICATION SECURITY			
LOGIN Input Required: username, password			
Login1	Perform the login to MacXim sending a message with a correct username and password	The login is correctly performed	
Login2	Perform the login to MacXim sending a message with an incorrect username and password	The system must return an error message	
LOGOUT Input Required: -			
Logout1	After Login1 test, check if the logout is properly performed	the logout is properly performed	
PROJECT MANAGEMENT			
PROJECT UPLOAD Input Required: project name, version, [release], [revision], repository type, url repository, [username], [password]			
ProjectUpload1	Upload project with the following parameters: project1, 1, svn, http://qualipso.dscpi.uninsubria.it/svn/svntest/ , svntest, svntest	The project is correctly uploaded	
ProjectUpload2	Upload the same project as ProjectUpload1	The system must return an error message	
GET PROJECT LIST Input Required: -			
GetProjectList1	After ProjectUpload1 test, check if the project is included in the list	the project is included in the list	
GET PROJECT METADATA Input Required: project name, version, [release], [revision]			
GetProjectMetadata1	After ProjectUpload1 test, check if metadata are correct	The project metadata are correct	
DELETE PROJECT Input Required: project name, version, [release], [revision]			
DeleteProject1	After ProjectUpload1 test, delete a project with the following parameters: project1, 1	The project is deleted	
DeleteProject2	After ProjectUpload1 test, delete a project with the following parameters: project1 (without specifying release number)	The system must return an error message	
PROJECT ANALYSIS general remarks: all tests will be executed on the application: http://qualipso.dscpi.uninsubria.it/svn/svntest/analysis/ , svntest, svntest			
GET ALL APPLICATION LEVEL METRICS			
ApplicationMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	

ApplicationMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
ApplicationMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	
GET ALL PACKAGE LEVEL METRICS			
PackageMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
PackageMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
PackageMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	
GET ALL CLASS LEVEL METRICS			
ClassMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
ClassMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
ClassMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	
GET ALL METHOD LEVEL METRICS			
MethodMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
MethodMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
MethodMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	

5.2.2 – Il Framework JUnit

Inserendo i test di unità all'interno del codice sorgente possono essere generati dei problemi, quali, ad esempio, il disordine generato durante la produzione del codice da testare, che si va a sommare a quello dei test e la presenza di un unico blocco di codice contenente test e applicazione, che non può essere diviso al momento della consegna all'utente finale.

Eventuali errori, nei test di questo tipo, sono frequentemente dovuti all'uso di membri privati o metodi, a cui l'interfaccia del modulo applicativo non avrebbe normalmente accesso e alla difficoltà nell'automatizzare i test, che non potranno essere eseguiti in modo indipendente.

Per ovviare a tali problemi si opera una separazione tra codice da testare e i relativi test, ottenendo così una loro indipendenza e automatizzazione.

Per ottenere ciò, è necessario un testing framework, ovvero una struttura di supporto su cui possono essere organizzati, progettati e valutati i test di unità.

E' possibile svilupparne uno proprio, comperarne uno esistente sul mercato oppure appoggiarsi a prodotti open source.

Si è scelto di mostrare l'utilizzo di Junit [y], un framework open source realizzato da Erich Gamma e Kent Beck nel 1997, che facilita la scrittura di test di unità scirtti in linguaggio Java.

I motivi che hanno portato a questa scelta sono: il risparmio di tempo per scrivere un testing framework proprio, il vantaggio di non doverlo comperare, la sua facile integrazione nel processo di compilazione e la disponibilità di materiali, esempi d'uso e affidabilità del prodotto, garantiti da un suo diffuso utilizzo presso la comunità Open Source.

E' un componente software costituito da varie classi e package, che deve essere usato secondo regole precise.

Tale framework aiuta ad organizzare e facilita le operazioni di testing, introducendo la possibilità di rappresentare casi di test, relativi input validi o non validi, risultati attesi e rendendo disponibili dei rapporti riassuntivi dei test, generati in modo automatico.

L'implementazione dei test di unità con JUnit è facile e veloce, consentendo una separazione tra codice sorgente e codice dei test,

aiutando a mantenere l'indipendenza tra un test e un altro e automatizzando il processo.

In questo modo c'è un notevole risparmio di tempo, che consente agli sviluppatori di dedicare maggiore attenzione alla qualità del software.

I test realizzati con JUnit, infatti, possono essere rieseguiti ogni volta che se ne presenti la necessità, senza dover riscrivere, modificare, osservare e analizzare manualmente.

Ogni modifica ai test non andrà a modificare il codice sorgente, permettendo al team di sviluppo di distribuire il prodotto senza dover cancellare i test o senza dover ripristinarli a seguito di modifiche al prodotto.

Grazie a JUnit, alla sua rapidità e semplicità, viene favorita la fusione della fase di progettazione e implementazione dei test in un unico passaggio.

Infatti, unitamente all'uso della T-Doc, che permette di avere una visione globale e completa dei test realizzati, risulta immediata la scrittura dell'implementazione di un caso di test.

Anche le procedure di esecuzione saranno uguali per tutti i tipi di test, dove verrà illustrato l'avvio e l'esecuzione dell'intera test suite, quindi l'intera collezione dei test.

L'implementazione, infatti, comincia solitamente con la creazione di un package, separato dal progetto, contenente tutto ciò che riguarda i test.

In java l'idea del test di unità è quella di associare ad ogni classe la relativa classe di test e al suo interno valutare ogni singolo metodo in funzione dei valori attesi.

Abitualmente tutte le classi di test e i loro relativi casi di test, vengono richiamati da una o più test suites, una struttura che consente di eseguire in modo pratico e veloce tutti i test di unità presenti.

L'implementazione di una classe di test prevede obbligatoriamente l'importazione del framework Junit e di tutte le classi del progetto, che si intendono testare.

A partire dalla versione 4 di JUnit si è introdotto l'uso delle annotazioni, novità di Java 5, che ha reso ancora più semplice e intuitivo lo sviluppo delle classi di test [x].

Nelle versioni precedenti era necessario conoscere la struttura delle classi JUnit da estendere e bisognava obbligatoriamente chiamare i casi di test come il metodo del progetto testato, preceduto dalla stringa "test", per permettere la loro esecuzione in automatico.

Tale esecuzione automatica avveniva grazie alla proprietà di "reflection" del linguaggio Java, caratteristica che permette di ottenere informazioni, durante l'esecuzione, su classi, interfacce, metodi e oggetti.

Grazie alle annotazioni ciò non è più necessario, in quanto i casi di test vengono identificati da appositi tag nel codice, che vengono valutati singolarmente durante la fase di esecuzione.

L'esecuzione dei test sarà perciò guidato dalla valutazione delle seguenti annotazioni nel codice sorgente:

ANNOTAZIONE	DESCRIZIONE
@Test	Identifica un metodo che rappresenta un caso di test, può essere parametrizzata (ad esempio: @Test(timeout=100))
@Ignore	Identifica un metodo di test da non eseguire, equivale al commento del codice di tutto il metodo
@Before	Indica un metodo che verrà eseguito prima di ogni caso di test (@Test) e può contenere l'inizializzazione di alcuni oggetti necessari alla sua esecuzione (ad esempio l'apertura di una connessione al database)
@After	Indica un metodo che verrà eseguito dopo ogni caso di test (@Test) e solitamente si occupa di chiudere le risorse che erano state aperte in precedenza (tramite @Before)
@BeforeClass	Si comporta come @Before, ma viene eseguito una sola volta come primo metodo della classe di test
@AfterClass	Si comporta come @After, ma viene eseguito una sola volta come ultimo metodo della classe di test

Un'altra importante caratteristica dei casi di test in JUnit è l'uso dei metodi "assert", che portano al fallimento del test nel caso in cui almeno uno di essi fallisca.

Essi sono metodi statici contenuti nella classe "org.junit.Assert" e possono essere usati direttamente (Assert.assertEquals(...)), oppure referenziati attraverso un'import statico all'inizio della classe (import static org.junit.Assert.*).

Generalmente, l'uso più comune di un metodo di assert, effettua una comparazione tra un risultato di esecuzione e il risultato atteso, ma vi sono diverse tipologie di metodi di assert.

Alla luce di tali conoscenze, è possibile implementare facilmente una classe di test, come mostrato nella [Figura 17], dove per migliorare la leggibilità e focalizzare l'attenzione su JUnit sono stati omessi i commenti T-Doc.

```
1  //package riservato ai test e separato dal codice sorgente
2  package project.testPackage;
3
4  //import statico di org.junit.Assert
5  import static org.junit.Assert.*;
6  //import del framework junit
7  import org.junit.*;
8  //import della classe del progetto da testare
9  import project.projectPackage.Class1;
10
11 //classe di test
12 public class TestClass1 {
13
14   //vari metodi da eseguire prima o dopo la classe
15   @BeforeClass
16   public static void initClass() {
17   }
18   @AfterClass
19   public static void endClass() {
20   }
21   //vari metodi da eseguire prima o dopo i casi di test
22   @Before
23   public void initMethod() {
24   }
```

```
25   @After
26   public void endMethod() {
27   }
28
29   //caso di test sul costruttore di Class1
30   @Test
31   public void testClass1() {
32     Class1 mySource = null;
33     assertNull(mySource);
34     mySource = new Class1();
35     assertNotNull(mySource);
36   }
37
38   //caso di test su method1 della classe Class1
39   @Test
40   public void testMethod1() throws IOException {
41     Class1 mySource = new Class1();
42     double result = mySource.method1();
43     double expected = 99;
44     assertFalse(result == 0);
45     assertTrue(result == 99);
46     assertEquals(expected, result);
47   }
48 }
```

Figura 17 - *Esempio di creazione di una semplice classe di test con JUnit.*

E' interessante notare come l'ordine di esecuzione dei metodi della classe di test sarà: initClass(), initMethod(), testClass1(), endMethod(), initMethod(), testMethod1(), endMethod() e infine endClass().
Quando si hanno a disposizione tutte le classi di test, è possibile procedere con l'implementazione della test suite, in modo da possedere una struttura che sia in grado di lanciare tutte queste classi.
Assegnare gruppi di test omogenei alla stessa test suite offre anche la possibilità di strutturare e ordinare i test in maniera più efficiente.
All'interno della classe designata a lanciare la test suite è sufficiente inserire, prima dell'apertura della classe stessa, due particolari annotazioni: @RunWith e @Suite.SuiteClasses, entrambe parametrizzabili.

```
1 //annotazione che indica l'esecuzione
2 //dei test tramite la classe Suite
3 @RunWith(Suite.class)
4 //annotazione che indica le classi di test
5 //richiamate dalla classe Suite
6 @Suite.SuiteClasses({
7    TestClass1.class
8 })
```

Figura 18 - *Esempio di annotazioni per la creazione di una test suite con JUnit.*

Come mostrato in figura i test verranno eseguiti dalla classe Suite, come specificato dalla parametrizzazione di @RunWith, mentre le classi di test richiamate da tale classe saranno specificate in @Suite.SuiteClasses, separate da virgole.
Una volta implementate le test suite questa fase può ritenersi conclusa.
Tuttavia, grazie alla generazione della documentazione T-Doc potranno emergere ulteriori test mancanti, quindi ancora da implementare.

5.2.3 – Il Progetto di Test

Dovendo procedere al testing di un applicativo come Macxim, realizzato da un team di sviluppo composto da diverse persone, si è ritenuto utile definire un'insieme di dati condiviso su cui potessero essere realizzati i test.
Come emerge dal piano di test e dall'analisi dei requisiti, Macxim ha come scopo principale quello di effettuare delle misurazioni sul codice applicativo.
I dati presi in input dai test saranno quindi estratti da un progetto o da un'applicazione, di cui si conoscono esattamente i corrispondenti valori delle metriche che dovrà calcolare Macxim.
E' stato stabilito l'uso del progetto "JavaSpaceJG" a tale scopo, in modo da evitare di avere diversi insiemi di dati di test disaggregati.

Inoltre, Macxim non lavora solo su piccole porzioni di codice, ma è in grado di misurare alcune caratteristiche dell'intero applicativo.
La definizione di un unico progetto di test consente di avere un riscontro delle abilità di Macxim in un ambiente di test sufficientemente grande e complesso, come nel caso di "JavaSpaceJG".
Inoltre, si è sfruttato il calcolo dei reali valori delle metriche su tale progetto una volta soltanto per tutti i test, non essendo così rapido e semplice.
Infatti, in un primo momento si è cercato l'appoggio di altri tool per la rilevazione di alcuni valori, come il plugin "metrics" per eclipse, ma non ritenendo il risultato soddisfacente ne in termini di correttezza, ne per una adeguata copertura di tutte le metriche presenti in Macxim, si è passati al calcolo manuale.
Di seguito è riportato il lavoro svolto per il calcolo di tutte le metriche, i risultati del calcolo manuale, i risultati prodotti dal plugin "metrics", dove presenti, e i valori calcolati da Macxim.

1			*****APPLICATION LEVEL***** for "JavaSpaceJG" test application				
2		METRIC NAME		MANUAL EXPECTED	METRICS EXPECTED	MACXIM VALUE	TOOL	DESCRIPTION
3	1	Num. Packages	TOTAL	3	3	3	Macxim	Number of package
4			MAX	X	X			
5			MIN	X	X			
6			STD-DEV	X	X			
7			MEDIAN	X	X			
8			AVG	X	X			
9	2	Num. Classes	TOTAL	14	14	14	Macxim	Number of classes
10			MAX	X	X			
11			MIN	X	X			
12			STD-DEV	X	X			
13			MEDIAN	X	X			
14			AVG	X	X			
...	...							
117	20	Num. Attributes Per Class	TOTAL	24	22	24	Macxim	Number of attributes
118			MAX	5	5	5		per class
119			MIN	0	0	0		
120			STD-DEV	1,622	1,545	1,622		
121			MEDIAN	1,5		1,5		
122			AVG	1,714	1,571	1 ,714		
123	21	Num. Of Public Attributes Per Class	TOTAL	1		1	Macxim	Number of public
124			MAX	1		1		attributes per class
125			MIN	0		0		
126			STD-DEV	0,257		0,257		
127			MEDIAN	0		0		
128			AVG	0,071		0,071		
129	22	Num. Effective Lines Of Code (eLOC)	TOTAL	417	544	417	Macxim	Number of effective
130			MAX	X	X			lines of code (eLOC),
131			MIN	X	X			that not includes
132			STD-DEV	X	X			blank lines, only-
133			MEDIAN	X	X			braces lines and only-
134			AVG	X	X			comment lines
135	23	Num. Effective Lines Of Code (eLOC) Per Class	TOTAL	393		393	Macxim	Number of effective
136			MAX	124		124		lines of code (eLOC)
137			MIN	5		5		per class, that not
138			STD-DEV	30,962		30, 962		includes blank lines,
139			MEDIAN	16		16		only-braces lines and
140			AVG	28,071		28,071		only-comment lines
...	...							

Figura 19 - *Valori delle metriche di Macxim calcolate sul progetto di test.*

Tale progetto, unitamente ai valori attesi per le metriche, è stato reso disponibile a tutti gli sviluppatori attraverso un repository centralizzato.

5.3 PROGETTAZIONE E IMPLEMENTAZIONE DEI TEST

5.3.1 – Macxim Test Suite

La progettazione e l'implementazione dei test sono avvenute in tempi rapidi, in relazione al quantitativo di lavoro da svolgere.

L'adozione del framework JUnit si è dimostrata un'ottima scelta, ma hanno giocato un ruolo fondamentale soprattutto l'accurata organizzazione dei package di test e la possibilità di inserire i commenti T-Doc all'interno del codice, che hanno consentito agli sviluppatori di definire test in modo standard e di inserire informazioni di progettazione all'interno del codice.

L'utilizzo della T-Doc è stato il vero stratagemma per fondere insieme, velocemente e con facilità, le fasi di progettazione ed implementazione dei test.

Le numerose e ripetute modifiche all'intera test suite sono state seguite da altrettante generazioni della T-Doc, che ha reso più facile e veloce le operazioni di check dei test, evidenziando mancanze di test su specifiche porzioni di codice e funzionalità.

Tutti i test realizzati sono stati organizzati in un unico progetto, separato dall'applicativo, che è stato chiamato "MacXim_1.0_Test".

All'interno di esso sono stati creati due package principali: "unit_test" e "acceptance_test", contenenti rispettivamente i test di unità e alcuni test di sistema previsti dal piano di test.

La realizzazione dei test di unità è iniziata con l'inserimento, nell'apposito package, dei test già realizzati dagli sviluppatori durante la produzione del codice sorgente, adeguatamente convertiti nello standard T-Doc.

Il lavoro è proseguito in modo regolare, con la creazione di test mancanti e con un'intensa attività di check dei test.

Infine, con l'aiuto della documentazione T-Doc, è stato possibile identificare subito dei test di integrazione, che sono stati subito implementati.

La realizzazione dei test di accettazione, invece, si è basata prevalentemente sulle tipologie di test definiti nel piano.

La maggior parte dei test sono stati concepiti da zero, prevedendo, attraverso una classe di supporto, una duplice modalità di esecuzione, tramite web service, oppure in locale.

Anche qui l'uso frequente e accurato di commenti T-Doc ha favorito l'organizzazione efficiente, soprattutto evidenziando il doppio significato dei test e le relative condizioni di successo e insuccesso sia nella modalità web service, che in locale [Figura 20].

```
1  /**
2  *Test check if parseGetProjectList() returns exactly the project uploaded
3  * parsing the request in WebService mode or Local mode
4  *
5  *@succeedIf parseGetProjectList() returns exactly the project uploaded
6  * parsing the request in WebService mode
7  *@failIf parseGetProjectList() doesn't return exactly the project uploaded
8  * parsing the request in WebService mode
9  *@succeedIf parseGetProjectList() returns exactly the project uploaded
10 * parsing the request in Local mode
11 *@failIf parseGetProjectList() doesn't return exactly the project uploaded
12 * parsing the request in Local mode
13 *@qualityAttribute functionality
14 *@scope system
15 *@author Jacopo Emoroso
16 *@version V1.1 - 22/09/09 - 16:00
17 *@see org.uninsubria.macxim.protocol.interpreter.parser.response.MessageParser
18 */
19 @Test
20 public void testGetProjectsList() {
21
22   String[] projectList =null;
23   try {
24     projectList =(String[]) testInit.getProjectsList(AllTests.isWebService);
25   } catch (Exception e) {
26     System.out.print("Error while parsing the message");
27     e.printStackTrace();
28   }
29   assertNotNull(projectList);
30   assertTrue(Arrays.binarySearch(projectList, AllTests.uploadProjectId)==1);
31 }
```

Figura 20 - *Esempio di commento T-Doc su un test con due possibilità per i tag @succeedIf e @failIf, poichè eseguibile sia in modalità web service, che in locale.*

5.4 T-DOC IN UN PROGETTO REALE

5.4.1 – Generazione della T-Doc per Macxim

Nonostante si sia implementato solo il primo layer del tool T-Doc, i benefici non si sono fatti attendere.

La generazione e l'aggiornamento automatico della documentazione hanno permesso di snellire notevolmente la fase di progettazione e implementazione dei casi di test.

Durante la scrittura stessa del codice dei test ci si è accorti della maggiore disponibilità di informazioni utili.

Quando poi si è generata la vera e propria documentazione, i vantaggi sono emersi con maggior forza e si sono riflettuti sull'intero team di sviluppo, rendendo disponibili i risultati su un repository condiviso.

La generazione della documentazione è potuta avvenire in qualsiasi momento, su richiesta di qualsiasi componente del gruppo di lavoro, sia personale incaricato al testing, che sviluppatori dell'applicativo.

Il documento generato includeva tutte le modifiche al progetto di test fino a quel momento caricate sul repository centrale dell'applicazione, sia quelle effettuate dal richiedente dell'aggiornamento della documentazione, che quelle introdotte da altri.

Per praticità si è scelto di procedere alla generazione sempre tramite l'interfaccia fornita da Eclipse, usato da tutti gli sviluppatori, secondo le modalità descritte nel paragrafo 4.3.2.

5.5 RISULTATI PRODOTTI DAL TESTING

5.5.1 – Risultati Ottenuti

Come prova empirica dei benefici illustrati precedentemente è possibile considerare i miglioramenti ottenuti in fase di testing del prodotto.

Ad esempio, nei mesi di Luglio, Agosto e Settembre 2009 c'è stato un intensa attività di testing inerente il calcolo delle metriche sul progetto di test "JavaSpaceJG".

Questi test hanno prodotti risultati diversificati nel tempo ed in continua evoluzione, poiché sono stati sviluppati contemporaneamente all'introduzione del calcolo di nuove metriche da parte di Macxim.

Per questo motivo risulta particolarmente interessante il dato rilevato per la correttezza nel calcolo delle metriche con granularità a livello applicativo [figura 1].

Nonostante nel corso di questi tre mesi il numero di tali metriche sia aumentato del 48%, passando da 19 a 28, il numero di valori errati è quasi sempre diminuito.

Il bilancio complessivo, al termine di questo periodo, evidenzia che il numero delle metriche errate e corrette grazie al testing è passato da 11 a zero.

Grazie alla T-Doc sono stati facilmente trovati i difetti dell'applicativo che hanno determinato gli 11 malfunzionamenti riscontrati.

La generazione automatica e immediata della documentazione, con uno sforzo così ridotto da parte degli sviluppatori, ha permesso di concentrarsi sui test, sull'individuazione dei difetti e sullo studio delle soluzioni, producendo risultati sorprendenti in tempi brevissimi.

E' particolarmente interessante considerare il periodo compreso tra il 22 Luglio e l'1 Agosto.

In soli 8 giorni il numero delle metriche calcolate da Macxim è cresciuto del 28%, passando da 22 a 28 e introducendo un gran numero di classi, metodi e query a database.

Il numero di metriche calcolate in modo errato, invece di crescere in modo proporzionale alla crescita del codice, si è abbassato del 60%, passando da 10 a 4.

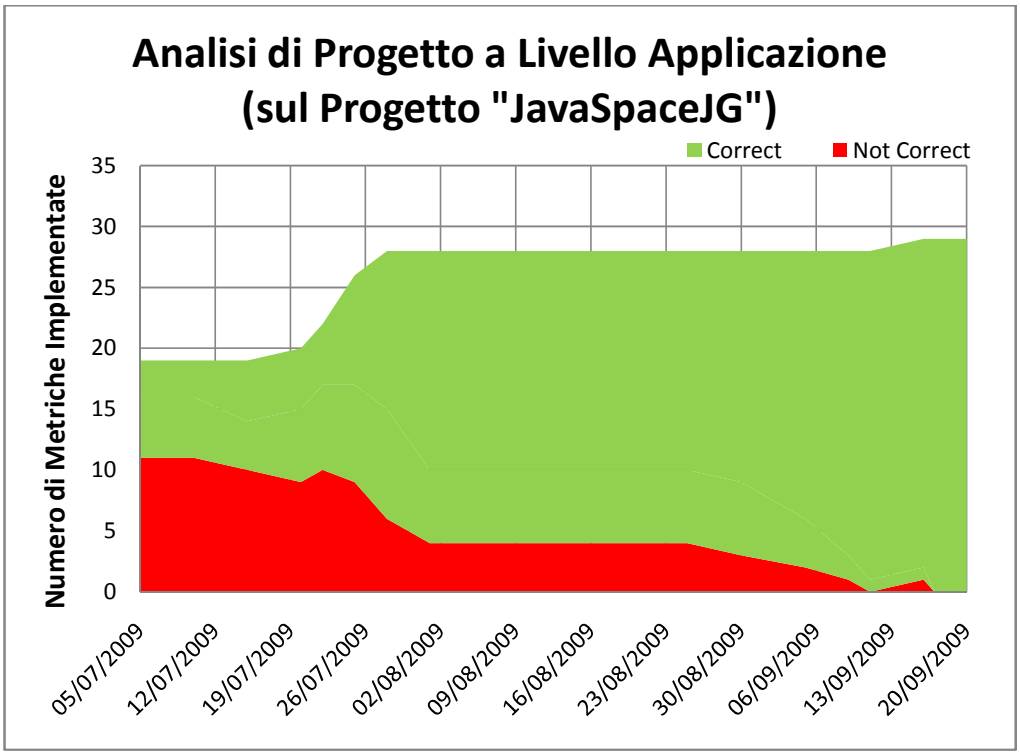

Figura 21 - *Dati ricavati dal testing di Accettazione per l'analisi di un progetto a livello di applicazione.*

Anche sugli altri livelli di granularità i risultati si attestano circa nella stessa misura.

Questo ci permette di concludere che la previsione dei benefici della generazione automatica della documentazione di test mediante T-Doc fosse corretta.

Inoltre, con il caso di studio Macxim, è stato possibile validare l'approccio introdotto con la T-Doc e conoscere meglio le sue potenzialità, gettando le basi per lo sviluppo di una versione stabile del tool, che possa essere utilizzata in progetti Open Source reali.

6.

CONCLUSIONI E SVILUPPI FUTURI

6.1 CONCLUSIONI

In questa tesi è stato proposto T-Doc, un tool che semplifica la generazione della documentazione di testing.

Attraverso lo studio dell'attività di testing, l'esame delle diverse problematiche e l'analisi di tool e framework esistenti, è emerso come T-Doc sia un prodotto unico nel suo genere ed in grado di rispondere adeguatamente alle esigenze di questa attività.

E' stato mostrato come T-Doc sia in grado di fornire un supporto alla generazione della documentazione durante l'intero processo di testing.

Il tool, infatti, si è dimostrato più che adeguato alla generazione della documentazione tecnica di tutti i casi di test in fase di progettazione ed

implementazione, al suggerimento per lo sviluppo di test di integrazione e regressione e alla raccolta e documentazione dei report sugli esiti dei casi di test.

E' stata definita e descritta l'architettura del tool e la sua capacità di memorizzare i dati elaborati, attraverso l'uso di un repository centrale dedicato, accessibile a tutto il gruppo di lavoro.

E' stato inoltre implementato il primo layer T-Doc, responsabile della generazione automatica della documentazione dei casi di test.

Si è mostrato come la T-Doc sia in grado di aiutare a fronteggiare le problematiche relative alla generazione e aggiornamento della documentazione di test, automatizzando il processo, aggregando i dati, favorendone la standardizzazione e la condivisione.

Inoltre, si è dimostrata la validità del tool creato in un contesto Open Source reale, il testing del progetto Macxim.

Ciò ha permesso di capire a fondo il funzionamento del primo layer della T-Doc durante la sua esecuzione, per la generazione della documentazione dei casi di test in fase di progettazione ed implementazione.

Sono stati compresi a fondo i benefici derivanti dall'utilizzo del tool ed è stato possibile quantificare le sue reali potenzialità.

Dal caso di studio Macxim, è emersa anche l'utilità e l'efficacia esprimibile attraverso i due layer in fase di implementazione, per il recupero e la documentazione di report di esecuzione dei test , e per il suggerimento di test di integrazione e regressione.

6.2 SVILUPPI FUTURI

Attualmente solo il primo layer del tool T-Doc, per la generazione automatica della documentazione dei casi di test, è stato completamente implementato.
Gli altri due layer sono ancora in fase di implementazione e verranno presto integrati nel tool.
Anche questi due layer verranno validati attraverso il progetto Open Source reale Macxim, per osservarne il funzionamento in fase di esecuzione, facendo emergere benefici introdotti e debolezze del tool.
Inoltre, si sta estendendo tale validazione ad ulteriori casi di studio, per garantire maggiore validità dei risultati e per trasformare il tool T-Doc in un tool di riferimento per il testing del Software Open Source.

BIBLIOGRAFIA

Principali fonti bibliografiche:

1. Hass e Anne Mette Janassen, “Guide to Advanced Software Testing”, Artech House Publishers, 3/2003, **ISBN 9781596932869**
2. Martin Pol, Ruud Teunissen, Erik van Veenendaal, "Software Testing: A Guide to the Tmap Approach", Addison-Wesley, 2002, **ISBN 0-201-74571-2**
3. Lu Luo, “Software Testing Techniques”, School of Computer Science, Carnegie Mellon University, **http://www.scribd.com/doc/17222992/Software-Testing-Techniquies**
4. Institute for Computer Sciences and Technology of the National Bureau of Standards, “ANSI/IEEE 829-1983 IEEE Standard for Software Test Documentation-Description”, 1983
5. Institute for Computer Sciences and Technology of the National Bureau of Standards, “ANSI/IEEE 1008-1987 IEEE Standard for Software Unit Testing-Description”, 1987
6. Petra Stefankova, “Software That Makes Software Better”, The Economist, 6/03/2008, **http://www.economist.com/sciencetechnology/tq/displayStory.cfm?story_id=10789417**
7. Luigi Cirillo, “Test Driven Development”, Mr.Webmaster, 3/12/2009, **http://www.mrwebmaster.it/java/articoli/test-driven-development_1116.html**
8. “Future Of Software Testing”, AppLabs, 7/7/2008, **http://www.applabs.com/internal/app_whitepaper_future_software_testing_1v001.pdf**
9. Jiantao Pan, “Software Testing”, Carnegie Mellon University, spring 1999, **http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/**
10. Michele Banci, “Software Testing”, Università degli Studi di Firenze, **http://www.dsi.unifi.it/~banci/SWTest.pdf**
11. Standard ISO 9000: 2000, Quality management systems
12. Paolo Tonella, “Analisi e Testing del Software”, Università degli Studi di Trento, 6/11/2009, **http://selab.fbk.eu/swat/course.pdf**
13. Piergiuliano Bossi, “Vademecum al testing automatico”, Stacktrace, 23/11/2009, **http://stacktrace.it/2009/11/vademecum-al-testing-automatico-1/**
14. Patrick Detler, “An Introduction to the Art of Unit Testing”, Zend, 3/12/2007, **http://devzone.zend.com/article/2772-An-Introduction-to-the-Art-of-Unit-Testing-in-PHP**
15. Gary McGraw, “Software Security Testing”, Cigital, Settembre - Ottobre 2004, **http://www.cigital.com/papers/download/bsi4-testing.pdf**
16. Ian Sommerville, “Software Documentation”, Lancaster University UK, 7/11/2001, **http://www.literateprogramming.com/documentation.pdf**
17. Harshad Oak, “The importance of the humble Javadoc”, TechRepublic, 4/9/2003, **http://articles.techrepublic.com.com/5100-10878_11-5066709.html**

18. QualiPSo Project, "Analysis of relevant open source projects", 24/11/2008, **www.qualipso.eu/node/84**
19. Sandro Morasca, Davide Taibi, Davide Tosi, Luigi Lavazza, "Automatic Generation of Testing Documentation for OSS Products via Built-in Test", Università degli Studi dell'Insubria, 2010
20. Chengying Mao, Yansheng Lu, Jinlong Zhang, "Regression Testing for Component-based Software via Built-in Test Design", Huazhong Univ. of Scie. & Tech, China, 15/11/2007
21. Sarah Smith, "Test Plan Document for RealEstate Application", 22/8/2005
22. Lars Vogel, "Unit testing with JUnit 4.x", Vogella, 6/2/2010, **http://www.vogella.de/articles/JUnit/article.html**

SITOGRAFIA

Principali fonti web:

a. Wikipedia (Italia), **http://it.wikipedia.org/wiki/Collaudo_del_software**, **http://it.wikipedia.org/wiki/Javadoc**
b. Wikipedia (Internazionale), **http://en.wikipedia.org/wiki/Software_testing**
c. Università degli Studi di Trento, Analisi e Testing del Software, **http://selab.fbk.eu/swat/program.ml?lang=it**
d. Università degli Studi del Sannio, Ingegneria del Software, **http://web.ing.unisannio.it/**
e. Università di Catania, Testing del Software, **http://www.dmi.unict.it/~tramonta/se/L16_IntroTest.pdf**
f. JavaPortal.it, articoli JUnit, Test di Unità e Testing del Software, **http://www.javaportal.it/**
g. JBoss, **www.jboss.org**
h. Sun Javadoc e API Specification, **http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/index.html**, **http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html**, **http://www.ics.uci.edu/~alspaugh/cls/shr/javadoc.html**
i. DoxyGen, **http://www.stack.nl/~dimitri/doxygen/**
j. Testopia, **www.mozilla.org/projects/testopia**
k. Fitness, **http://fitnesse.org**
l. Eclipse TPTP, **www.eclipse.org/tptp/**
m. Salome-TMF, **https://wiki.objectweb.org/salome-tmf/**
n. Sun Javadoc Doclet, **http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/overview.html**
o. Sun Javadoc Taglet, **http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/taglet/overview.html**
p. Eclipse Project, **http://www.eclipse.org/**
q. Call Graph Tool, **www.certiv.net/projects/**
r. MacXim Project, **http://qualipso.dscpi.uninsubria.it/macxim/**
s. eXist DataBase, **http://exist.sourceforge.net/**
t. PMD Tool, **http://pmd.sourceforge.net/**
u. CheckStyle Tool, **http://checkstyle.sourceforge.net/**
v. FindBugs Tool, **http://findbugs.sourceforge.net/**
w. Spago4Q Project, **www.spago4q.com**
x. HTML.IT, articoli Java e Testing del Software, **http://java.html.it/articoli/leggi/2831/junit-40-testing-di-unapplicazione-java/** **http://java.html.it/articoli/leggi/2831/junit-4-annotation-vs-reflection/2/**
y. JUnit, **www.junit.org**

APPENDICE 1: MACXIM T-DOC