L'affidabilità del software

Keywords

Software, fault, reliability.

Keywords - native language

Programma, errore, affidabilità.

Author
Nicola Zicari
Last modified by
Nicola Zicari

Language
Italian

Peer Reviewers
Giuseppe La Russa
Maria La Gioia

Created on
2019-01-10
Last modified on
2019-01-13
Submission date
2019-01-10

Status

Rated

Attachments

No attachments.

Reviewer 1

Reviewer 2

Title (in English)

Software reliability

Abstract

L’affidabilità è una caratteristica intrinseca del software e come tale deve essere definita in termini di requisiti, progettata adeguatamente, sviluppata coerentemente con il disegno e valutata lungo l’intero ciclo di vita.

Il funzionamento corretto e prolungato del software incorporato (embedded) nei sistemi di cui ci si serve quotidianamente è di estrema importanza per i milioni di utenti che li utilizzano (malati sottoposti a terapie particolari, passeggeri dei voli aerei e dei treni, automobilisti, clienti di ipermercati, banche, ecc.).

E’ necessario quindi prestare estrema attenzione a tutte le fasi del ciclo di vita di un software per evitare malfunzionamenti imprevedibili che vanno a discapito degli utenti e che possono avere conseguenze anche molto gravi.

Abstract (in English)

The Reliability is an intrinsic feature of the software and as such it has to be defined in terms of requirements, planned appropriately, developed in accordance with the design and assessed during all the lifecycle.

The proper and extended functioning of the embedded software in systems used daily is extremely important for millions of users (patients treated with special therapies, passengers travelling by plane or train, drivers, hypermarket customers, banks, etc.)

Then it is necessary to pay close attention to all the stages of the lifecycle of a software to prevent unpredictable failures that can damage customers and can brought serious consequences.

Introduction

Lo sviluppo del software è un'attività complessa ma spesso le problematiche che esso comporta vengono affrontate con superficialità e nell'immaginario collettivo si pensa che chiunque abbia un po' di conoscenze informatiche sia in grado di realizzare sistemi software complessi in breve tempo.

In relatà realizzare software affidabile è invece un compito arduo e complicato che richiede notevole professionalità e utilizzo di metodi e strumenti adeguati.

Occorre quindi formare adeguatamente i futuri programmatori, facendogli comprendere l'importanza e la difficoltà del lavoro che li attende.

Sarà quindi necessario che comprendano concetti come affidabilità, robustezza, integrità. In particolare, ritengo sia opportuno insistere sull'affidabilità, perchè è la caratteristica che maggiormente interessa all'utilizzatore finale.

La Reliability (Affidabilità), è definita come la probabilità che un sistema funzioni correttamente in un intervallo di tempo di durata prestabilita, nell'ipotesi che esso sia operato e manutenuto come prescritto dalle procedure operative e, rispettivamente, di manutenzione. In altre parole la R(t) è la probabilità che il sistema, correttamente operato e manutenuto, non abbia avarie nell'intervallo di tempo [0, t].

Il fallimento di molti progetti software (nel campo aerospaziale e non solo) è attribuibile alla mancata, insufficiente o non corretta:

- Definizione dei requisiti di affidabilità (e di sicurezza) e relativa specificazione
- progettazione dei requisiti di affidabilità nel software
- verifica e validazione della corretta definizione, progettazione e implementazione di tali requisiti.

Durante il ciclo di vita di un software, le modifiche sono inevitabili (mutate esigenze del committente, adeguamento a nuove norme di legge etc), e ogni variazione di codice compromette la stabilità del programma. E' quindi indispensabile che nella fase di progettazione si preveda la possibilità di effettuare facilmente le modifiche e che il lavoro finale risulti facilmente testabile sia nella parte modificata sia nelle parti adiacenti.

La progettazione di un software affidabile deve inoltre considerare l'integrazione perfetta dell'architettura software in quella hardware e la capacità del sistema di riconoscere gli errori, gestirli e intercettare condizioni imprevedibili.

Content

Tutte le applicazioni software sono descritte da un insieme di **specifiche** (scritte in modo testuale oppure usando una metodologia più formale) che definiscono il comportamento richiesto all'applicazione. Ogni punto della specifica (che definisce quindi un aspetto puntuale del sistema) è detto **requisito**.

Possiamo definire l'affidabilità di un'applicazione software come segue (Randell, 1978):

L'affidabilità è una misura della capacità di un sistema di comportarsi come stabilito dalle sue specifiche.

E’ importante notare che le suddette specifiche, oltre ad essere complete, consistenti e non ambigue (qualità tutt’altro che scontate) devono anche definire i **tempi di risposta** del sistema ai vari eventi.

Introduciamo ora il concetto di **failure** (basato ancora su [Randell, 1978]):

Si parla di failure quando il comportamento di un sistema differisce da quanto stabilito dalle sue specifiche.

Notare come la failure sia riferita al comportamento esteriore del sistema: una failure è causata da dei problemi interni all’applicazione, che alla fine si manifestano in comportamenti esteriori scorretti. Questi problemi interni sono chiamati **fault**.

Possiamo classificare i fault in funzione della loro origine (Burns, 1989):

1. Fault causati da errori nelle specifiche. Questa è la caratteristica più specifica di un’applicazione software e si ritiene anche ([Levenson 1986]) che sia la causa principale delle failure software.
2. Fault introdotti da errori nel disegno o nell’implementazione.
3. Fault introdotti da guasti nei processori su cui viene eseguita l’applicazione software.
4. Fault introdotti da guasti (transitori o permanenti) nel sistema di comunicazione.

Tutte le tecniche affidabilistiche software ruotano attorno alla identificazione delle failures ed alla eliminazione dei Fault.

I metodi disponibili per minimizzare la presenza di Fault e delle loro conseguenze sono sostanzialmente due:

1. Evitare il più possibile che dei Fault vengano introdotti nel sistema, o comunque eliminarli prima che il sistema venga messo in esercizio (**fault prevention**). Questi metodi si applicano principalmente alla risoluzione di Fault di tipo (1) e (2).
2. Costruire il sistema in modo che sia capace di sopportare la presenza di Fault quando questi si verificheranno (**fault tollerance**). Questi metodi si applicano principalmente alla risoluzione di Fault di tipo (3) e (4).

Ovviamente i due approcci non sono esclusivi, ma complementari: si cercherà di eliminare il più possibile i Fault in fase di progettazione, ma si inseriranno anche dei meccanismi di protezione all’interno dell’applicazione che permettano di garantire un corretto funzionamento anche quando dei Fault (che non saremo riusciti ad eliminare in fase di progettazione) si manifesteranno. Tutte le tecniche di fault prevention si riassumono nell'adozione di una precisa metodologia di sviluppo del software o, come si dice in termini tecnici, nell'adozione di un preciso **ciclo di vita del software**, da seguire in modo sistematico.

Forniremo qui di seguito un'introduzione a tali tecniche, per quanto di interesse allo sviluppo di un'applicazione software real-time. Possiamo dire che, indipendentemente dal tipo di applicazione da sviluppare, dai linguaggi di programmazione e dagli ambienti di sviluppo scelti, le fasi principali seguite in un qualsiasi progetto software sono le seguenti:

1. **Definizione dei requisiti**: capire il più esattamente possibile che cosa l’applicazione deve fare.
2. **Disegno**: identificare i vari moduli, le loro funzioni e le loro interazioni (il modo in cui i moduli vengono identificati e definiti varia pesantemente a seconda della metodologia

scelta: top-down, a oggetti, ecc., ma in questo contesto tali differenze non sono significative).
3. **Implementazione** dei singoli moduli (inclusiva della verifica di corretto funzionamento di ogni modulo).
4. **Integrazione** dei moduli (inclusiva della verifica di corretta comunicazione tra i moduli).
5. **Verifica** del corretto funzionamento (rispetto ai requisiti iniziali) dell'applicazione.

Una scarsa attenzione ad uno qualunque dei punti del ciclo di vita di un software, può portare ad errori anche gravi del sistema progettato.

Un caso tristemente emblematico è quello del THERAC-25, un dispositivo computerizzato per la terapia a emissione di radiazioni destinata ai malati di cancro. A causa di un errore nel software di controllo, alcune di queste macchine rilasciarono dosi eccessive di radiazioni uccidendo alcuni pazienti e menomandone altri (negli anni 1985-1987). Secondo la versione ufficiale:

- il programma fu scritto da un singolo programmatore non più rintracciabile e le cui qualifiche e livello di studi non furono mai accertati
- l'elenco di specifiche non fu ritrovato
- il programma era scarsamente documentato
- il piano di collaudo non esisteva
- gli errori erano dovuti a una soluzione ad hoc (dilettantistica) delle attività multitasking (controlli della tastiera, schermo, stampante e rilascio di radiazioni)
- anche il software del modello precedente (THERAC-20) conteneva gli stessi errori, ma era dotato di blocchi hardware che impedivano meccanicamente il rilascio di radiazioni in dosi eccessive (nel THERAC-25 erano stati sostituiti da controlli software)
- Come esempi concreti di quanto affermato, possiamo fare riferimento a quanto accaduto durante il lancio inaugurale del razzo Ariane IV, dove per un problema software, dopo pochi minuti dal lancio il razzo esplose. Si scoprì che la mancanza di ridondanza software (un doppio controllo da parte del programma), dovuto ad una carenza delle specifiche (si ritorna al ciclo di vita del software), aveva fatto si che un sensore non potesse funzionare correttamente.

In sistemi software complessi vengono previste anche situazioni estremamente improbabili: poiché anche l'evento più improbabile prima o poi si verifica è sempre bene accertarsi che tutti questi casi siano adeguatamente testati in modo da non far fallire il sistema quando si verificano.

Il 15 gennaio 1990 per un banale errore di programmazione in una parte non adeguatamente testata, le comunicazioni a lunga distanza della compagnia telefonica AT&T vennero bloccate per 9 ore, tempo necessario ai tecnici per individuare il guasto. Migliaia di utenti vennero lasciati senza possibilità di telefonare e circa 70 milioni di telefonate non poterono essere effettuate. Oltre ad un grave danno economico, l'azienda AT&T ebbe una grave ricaduta nell'immagine in quanto stava portando avanti un'aggressiva campagna pubblicitaria sulla affidabilità del suo servizio rispetto alle altre compagnie sue concorrenti

Nel 1999 il Mars Climate Orbiter si è distrutto nell'atmosfera di Marte dopo un viaggio di 2 anni e dopo uno spreco di vari milioni di dollari da parte della NASA ed dell'ESA, finanziatrici e realizzatrici del progetto.

Il fallimento fu dovuto al fatto che nello scrivere alcuni parti del codice si considerarono alcuni dati come espressi in unità di misura inglesi quando invece erano espressi in unità metrica-decimale. Infatti il progetto era stato sviluppato parte in Europa, dove si utilizza l'unità metrico-decimale, e parte in USA, dove si utilizzano le unità di misura inglesi, ed in ognuno dei due Paesi i tecnici davano per scontato che l'altro usasse la propria unità di misura.

Risulta quindi fondamentale per evitare malfunzionamenti, prevedere un adeguato test del programma.

Il testing o collaudo è il procedimento che verifica la correttezza, la completezza e l'affidabilità di un software.

Assicurare la qualità del prodotto tramite la ricerca e correzione dei difetti è quindi l'obiettivo fondamentale di questa procedura. Solo tramite il testing possiamo avere la certezza che il software corrisponda esattamente ai requisiti definiti in fase di analisi delle specifiche tecniche.

E' fondamentale che chi programma (o si appresta ad intraprendere questa attività) abbia ben chiara l'importanza di questa fase del ciclo di vita di un software. Posso aver scritto il programma più complesso e funzionale del mondo, ma se non sono sicuro del suo corretto funzionamento non dovrò mai rilasciarlo.

A prescindere dalla metodologia utilizzata, va definito un processo fondamentale suddiviso in step.

1. Elaborazione di strategia e piano di test: oltre alla definizione di tutte le caratteristiche e le funzionalità da testare, è necessario individuare un approccio, determinare gli ambienti e pianificare il flusso. Il piano test comprende le attività per i progetti di sviluppo e manutenzione che definiscono gli obiettivi del test. La pianificazione è influenzata da molteplici fattori: dalla disponibilità di risorse, dalla politica di test e dalla sua strategia, dai cicli di vita e dai metodi di sviluppo, dai vincoli e dalla criticità del contesto.
2. Design del test: in questa fase si realizza la suite di test, ossia l'insieme di casi d'uso necessari per la validazione. Durante il test design, vengono elaborate condizioni o variabili sotto le quali un tester determina se una applicazione o sistema software risponde correttamente o meno.
3. Esecuzione del test: Durante l'esecuzione del test, le suite di test vengono elaborate in base alla programmazione del test.
4. Conclusione del test: Nella conclusione dei test vengono raccolti i dati relativi alle attività di test, testware e qualsiasi altra informazione pertinente.

Vanno inoltre illustrate le varie tipologie di test, mettendo in evidenza per ciascuna di esse quali sono i vantaggi e quali gli svantaggi.

Tra queste abbiamo:

- Functional Testing: prevede dei test che valutano le funzioni che il sistema deve eseguire. Le funzioni sono le azioni che dovrebbe fare il sistema.
- Non-functional Testing: Valuta le caratteristiche di software e di sistemi come l'usabilità, l'efficienza o la sicurezza.

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. 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.

Affinchè quanto illustrato non resti una pura trattazione teorica, vanno predisposte numerose attività in laboratorio nel corso delle quali, si dovranno affrontare tutte le fasi del ciclo di vita di un software. Si organizzeranno vari gruppi di lavoro ognuno con un compito diverso (analisi del problema, disegno del vari moduli di programma, scrittura del programma, testing) e si pianificheranno opportune batterie di test osservando per ognuna di esse le risposte del programma e valutando gli eventuali danni causati dai possibili errori.

Una possibile proposta di lavoro, abbastanza semplice da poter essere utilizzata in maniera agevole, è la realizzazione di un sistema di prenotazione voli online. Una volta stabilita la composizione dei gruppi ed i relativi compiti, si illustreranno le problematiche principali che si possono incontrare, si mostreranno possibili soluzioni e si darà il via alla fase esecutiva del progetto.

Pur essendo le fasi di lavoro sequenziali tra loro, verranno comunque progettate le batterie di test, assicurandosi di prevedere tutte le possibili criticità delle varie procedure e tutti i casi particolari che si possono verificare, in maniera tale da poter rilasciare un software quanto più possibile affidabile.

Conclusions

A conclusione di quanto descritto, si può affermare che non solo i malfunzionamenti hardware possono generare gravi problemi di efficienza del sistema, ma anche e soprattutto bug del software portano a gravi conseguenze, che sono ancora meno tollerabili perché spesso causate dalla mancanza di professionalità di chi il software lo ha sviluppato e testato. Occorre formare adeguatamente tutti i partecipanti ad un progetto insistendo sull’importanza di ognuna delle fasi di un ciclo di vita di un software.

Un’analisi accurata dei requisiti genererà probabilmente delle specifiche ben dettagliate che faciliteranno i programmatori nel loro lavoro. Fondamentale è poi l’integrazione dei vari moduli che compongono il programma e la verifica puntuale dei risultati attesi. Il tempo speso nella fase di testing verrà recuperato in seguito, durante la fase di manutenzione del software e si eviteranno comportamenti imprevedibili che potrebbero avere conseguenze catastrofiche.

L’importanza del corretto funzionamento di applicazioni da cui ormai dipende la maggior parte delle attività strategiche della società moderna (banche, ospedali, etc.) richiede che la formazione di tutti i partecipanti ad un progetto software debba essere di livello adeguato.

Purtroppo in passato è stata sottovalutata l’importanza della formazione e sono stati affidati progetti importanti a personale non adeguatamente preparato con conseguenze in alcuni casi catastrofiche.

Scuola, università ed enti di formazione devono avere il compito di formare specialisti preparati e competenti che uniscano ad una solida conoscenza teorica delle problematiche anche la capacità di mettere in pratica quanto appreso. Non è più tempo di costruirsi “sul campo” la propria professionalità, il mercato ha bisogno di personale preparato che sappia da subito inserirsi in un gruppo di lavoro.

References

1. B. Randell, P.A. Lee, P.C. Treleaven (1978). Reliability Issues in Computing System Design. ACM Computing Survey Vol. 10 Issue 2 pp. 123-165
2. B. Randell (2007). A computer Scientist's reaction to NPfiT. Available from http://homepages.cs.ncl.ac.uk/brian.randell/Papers-Articles/JIT-Randell
3. M. L. Shooman (1990). Probabilistic Reliability: An Engineering Approach. 2nd Edition . Robert E. Krieger Publishing Company, Malabar, Florida.
4. J. D. Musa, A. Iannino, K. Okumoto (1990): Software Reliability Measurement, Prediction, Application. McGraw-Hill Publishing Company
5. L. H. Putnam, W, Myers (1992): Measures for Excellence – Reliable Software on Time, within Budget. Yourdon Press, Upper Saddle River, New Jersey 07458. Rome Laboratory.
6. B.Wade Rose (1994). FatalDose-Radiation Deaths linked to AECL Computer Errors. Available from http://www.ccnr.org/fatal_dose.html
7. A. Cristalli. 15 Gennaio 1990 AT&T crash. Available from http://www.grayhats.org/articles/52-hacking/249-15-gennaio-1990-atat-crach
8. Air Force Systems Command. Griffiss Air Force Base. NY 13441-5700. RL-TR-92-52 (1992)– Software Reliability, Measurement, And Testing – Software Reliability and Test Integration, vol. I and II.
   available from http://www.qualitaonline.it/qualitawp/affidabilita-del-software/
9. Quix.it available from https://www.quix.it/blog/collaudo/7119228