Sicurezza e Affidabilità

Adrian Castro

Anno scolastico 2018-2019

Contents

1 Introduzione

1.1 Testing

Osservando la storia più recente, è noto che gli strumenti utilizzabili (linguaggi di programmazione, strumenti di supporto allo sviluppo, strumenti di convalida di processi software) hanno subito una forte innovazione, che ha portato ad uno sviluppo più facile dei software. Più facile, ma comunque soggetto all'errore umano.

Nel mondo in cui viviamo, si tende a ragionare con elementi lineari. Ad esempio, se si esegue su un test di carico di un ponte, il risultato sarà una funzione continua: il ponte reggerà fino ad un certo carico. Non è possibile ragionare così quando si parla di *software testing*: non è detto che dei test effettuati con dei certi valori diano lo stesso risultato di altri test con altri valori. Non è quindi possibile determinare un intervallo in cui si ha la certezza che il software funzioni. Oltretutto, il software deve anche soddisfare i requisiti di progetto, tempi di risposta ragionevoli, sicuro, etc.

Un software, quindi, possiede delle caratteristiche che rendono il problema della qualità particolarmente difficile da risolvere:

- Requisiti di qualità:
  - Funzionalità;
  - qualità non funzionali (prestazioni, tolleranza ai guasti, sicurezza);
  - qualità interne (manutenibilità, portabilità);
- Struttura in evoluzione (e deterioramento);
- non linearità intrinseca (la funzione che descrive il suo comportamento in base agli input non è continua)
- distribuzione non uniforme dei guasti

1.2 Cybersecurity

Si vuole garantire nel software rilasciato delle protezioni rispetto a vari tipi di elementi. È necessario quindi, oltre a poter dimostrare che il software funziona, controllare se il servizio (o dati forniti) sono o non sono accessibili, se devono o non devono esserlo. In alcuni casi si vuole persino essere in grado di sapere **quando** un servizio (o un dato) deve essere accessibile (solo in determinate ore del giorno, solo dopo certi eventi, etc...), come può essere accessibile (se ci sono utenti con privilegi diversi e che possono quindi svolgere operazioni diverse), sapere chi può accedere al servizio e chi no, etc. Tutte queste problematiche, che stanno attorno alla vera funzionalità del sistema, vengono definite **problemi di sicurezza**.

Pertanto, quando un software:

- deve soddisfare un particolare requisito;
- deve essere accessibile solo da qualcuno in particolare;
- deve essere accessibile solo in particolari momenti e con certe modalità;

- deve essere accessibile solamente con certe opzioni;

si parla di **cybersecurity**.

Lo scopo di questa disciplina è di proteggere il sistema da possibili attacchi mirati. Il contesto è quindi dinamico: infatti, se la qualità non è quella attesa, è più facile per gli hacker violare il sistema ed attaccarlo. Qualità e sicurezza devono andare quindi di pari passo.

2 Testing e Verifica dell'Adeguatezza

Il termine **testing** si riferisce all'eseguire un programma ed osservarne i risultati, andandoli a confrontare con un risultato atteso (si devono quindi avere delle specifiche). Bisogna inoltre definire quante volte è necessario eseguire l'operazione di testing (*testing adequacy*). Un'altra, meno usata, definizione di testing si chiama "exhaustive testing", cioè che il sistema viene testato per tutti i possibili input. Vien da sè capire che questo tipo di testing è controverso, poiché non è possibile controllare tutti i possibili input (che sono infiniti, o comunque enormi). Questa controversia è riconducibile all'*halting problem*: può esistere un algoritmo che risponde *si* o *no* rispetto alla terminazione di un programma qualsiasi? *No.*

Quello che si ottiene attraverso la fase di testing è un'approssimazione ottimistica della qualità, ossia attraverso il test viene scelto un numero piccolo di tutti i possibili input (scelti in maniera intelligente). In particolare, si può ottenere:

- l'identificazione di un bug;
- le prove non manifestano bug (è ottimistico concludere che il programma funziona per tutti gli altri input)

2.1 Terminologia dei bug (IEEE)

Il termine *bug* ha varie sfumature:

- failure;
  osservazione del problema, ma non il codice che l'ha causato. In questa fase, ancora non si sa come correggere il problema
- fault;
  si analizza il malfunzionamento, si analizza il codice, si evidenzia il problema e si comprende come correggerlo
- error;
  ulteriore analisi del fault: ci si chiede il perché si è verificato l'errore, e di conseguenza ci permette di classificare i fault in base all'area in cui appartengono (es. errori di gestione della memoria, etc...)

2.2 Selezionare il test a partire da una failure

Come abbiamo detto in precedenza, non è possibile testare il software con tutti i possibili input. Come prima cosa, si fanno delle prove che vanno a stimolare il sistema in maniera mirata per trovare

delle failure. Questo primo obiettivo è abbastanza intuitivo. Come seconda cosa, mano a mano che vengono eseguiti i test, che vengono identificati bug (e corretti), i pochi test rimasti da fare vano in convergenza. In pratica, una volta verificato che il software supera tutti i test previsti, ci si chiede *"cosa sappiamo?"* In particolare, ci chiediamo se i test che abbiamo fatto siano sufficienti per poter definire il software di qualità e pronto al rilascio.

Le fasi, riassumendo, sono due:

1. effettuare prove intelligenti mirate alla risoluzione dei bug;
2. verificare che le prove scelte nel punto precedente siano anche adeguate.

Esempio 2.1

Consideriamo la seguente applicazione sulla quale vogliamo eseguire dei test:

Figure 1: Simple Game

L'applicazione è dotata di due campi di testo (numerici o testuali) ed un pulsante per far partire il sistema. È bene sottolineare che non si vuole entrare nei dettagli specifici del programma, ma semplicemente si vuole lavorare sula base dell'interfaccia data.
Il programma, per avviare il gioco, richiede di specificare il numero di giocatori ed il nome della partita. Il tasto "Start game" deve avviare il gioco in maniera corretta.
Effettuare il testo di questa applicazione vuol dire inserire dei valori nei campi specificati, premere "Start game" ed osservare il comportamento in modo da verificarne la correttezza.
Si richiede di scegliere al più *dieci* prove da effettuare:

Caso standard	Casi di errore previsto	Casi di stress
Players ≥ 1	Nome con caratteri particolari potrebbero mandare in crash il sistema	Players molto elevato
	Players con lettere invece di numeri	Nome stringa lunga
	Players ¡ 0	
	Campi vuoti	
	Players è un double	

Si può osservare che si sono considerati i casi in cui il sistema potrebbe andare in crash (lo sviluppatore non è stato attento).

2.3 Test Obligations and Test Cases

Quando si ragiona sull'identificazione dei test, vengono identificati una serie di elementi (come nell'esempio sopra) che vengono indicati come "test objectives" o "test obligations".

Se non si conosce la specifica del sistema, si possono individuare casi di crash/malfunzionamento che si verificano per qualunque sistema, indipendentemente dal dominio applicativo. Tuttavia, molti fallimenti dipendono dal sistema, pertanto, quando si passa all'obiettivo del test alla corrispondente implementazione, si ha che quest'ultima è fatta dai risultati attesi degli stessi. Per conoscere questi risultati attesi, è *necessario* conoscere la specifica del sistema.

- **Test Obligations**: test che devono essere effettuati almeno una volta per verificare il corretto funzionamento dell'applicazione. Non è necessario conoscere la specifica;
- **Test Case**: test fatti ad-hoc per dei casi particolari. È necessario conoscere la specifica *prima* di eseguire il test.

Esempio 2.2

> L'applicazione (*Simple Game*) permette all'utente di creare delle istanze di un gioco. Per creare tali istanze è necessario specificare il numero di giocatori e un nome della partita. Il gioco richiede almeno cinque giocatori ed al massimo può essere giocata da trenta giocatori. Se i giocatori sono più di trenta, allora il gioco parte comunque, ma in una modalità specifica denominata "Team-Mode". Il gioco può partire in modalità "Private-Mode" se il nome del gioco comincia con l'asterisco, oppure in "Public-Mode" se non comincia con l'asterisco (quindi in tutti gli altri casi).

Questo è un esempio di specifica della nostra applicazione. È pertanto possibile definire dei valori per il valore atteso di ciascun obiettivo.
Supponiamo di considerare l'input seguente:

- Players = 5
- Name = "nome"

Secondo la specifica, è possibile dedurre i valori attesi:

- Con 5 giocatori il gioco può partire in modalità normale
- Con "nome" il gioco partirà in "Public-Mode" in quant il nome non contiene un asterisco

Chiaramente, se il valore atteso corrisponde a quello del valore ottenuto con l'input specifi-

cato, allora il test ha avuto successo, altrimenti avremo un *bug*.
Quindi, alla luce della specifica sopra introdotta, è facile vedere come gli obiettivi di test prima individuati non siano più ottimali.

Caso standard	Casi di errore previsto	Casi di stress
Players ≥ 5	Campi vuoti	Players molto elevato
Players ≥ 5 e ≤ 30	Players < 5	
Nome con “*”		
Nome senza “*”		
Nome = “*”		
Nome con “*” in mezzo		
Nome con “*” e spazio		

Lo stesso obiettivo di test può essere raggiunto da più valori (considerando l'esempio precedente, è possibile provare l'applicazione per Players = 40, Players = 32, etc...).

3

4

5