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)

Un *criterio di adeguatezza* dovrebbe permettere di rispondere "si" o "no" alla domanda "L'insieme di casi di test progettati è sufficiente?" Può quindi essere visto come un predicato che è *vero* (soddisfatto) o *falso* (non soddisfatto) di una coppia del tipo `<programma, test suite>`.

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 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:

Figure 2: Casi per Simple Game

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 e 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 specificato, 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.

Figure 3: Casi per Simple Game dopo la specifica

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...).

2.4 Boundary Testing

Fino ad ora non abbiamo però tenuto conto dei cosiddetti *casi limite* o *boundary testing*, ossia i test effettuati usando valori che soddisfano una casistica molto specifica, che nel nostro esempio per "Simple Game" possono essere Players = 5, Players = 30, Players = 31.

La specifica distingue una serie di comportamenti, di cui tre risultano essere particolarmente di interesse:

- Players = 5
  Il gioco non funziona
- Players = 30
  Il gioco funziona in modalità non a squadre
- Players = 31
  Il gioco funziona in "Team-Mode"

Questi comportamenti dipendono dal parametro *numero di giocatori*, tuttavia, sono adiacenti l'uno all'altro (c'è un valore specifico per il quale ad un certo punto si passa da un comportamento ad un altro). Il comportamento del sistema è quindi *non-lineare*.

In generale quindi, si cercano di identificare tre cose:

1. Comportamenti normali
   Esplicitamente definiti dalla specifica

2. Comportamenti eccezionali
   Implicitamente suggeriti dalla specifica, e sono di "contorno" ai comportamenti normali
3. Casi di confine

I comportamenti individuati nel nostro sistema di esempio sono:

1. Comportamenti normali
   - Players ¡ 5
   - Players ≥ 5 e ≤ 30
   - Players ¿ 30
2. Comportamenti eccezionali
   - Players è una stringa (dovrebbe essere un numero)
   - Name è una stringa vuota (campo non compilato)
3. Casi di confine
   - Players = 4
   - Players = 5
   - Players = 30
   - Players = 31

3 Test Funzionali

Un primo tipo di *criteri di adeguatezza* di interesse sono i *test funzionali*, che prendono questo nome in quanto il criterio di adeguatezza viene definito analizzando "manualmente" la specifica e trasformandola in un elenco di test obligations.

Tipicamente, tutti i criteri di adeguatezza di tipo funzionale (e quindi basati sulla specifica) cercano di riprodurre la specifica come una lista di comportamenti rispetto ai quali è possibile identificare:

- comportamenti normali del sistema;
- comportamenti eccezionali;
- casi di confine.

In generale:

Figure 4: Tipi di approccio

Approccio Sistematico	Approccio Random
• Cerca di fare delle scelte specifiche guidate da dei principi di analisi della specifica • È legato alla persona che compie le scelte	• È più automatico • Evita che la scelta dei casi di test sia basata sull'abilità specifica della persona che esegue l'analisi • Sceglie una distribuzione per gli input e campiona automaticamente

Chiaramente, selezionare il tipo di approccio da usare di volta in volta è essenziale.

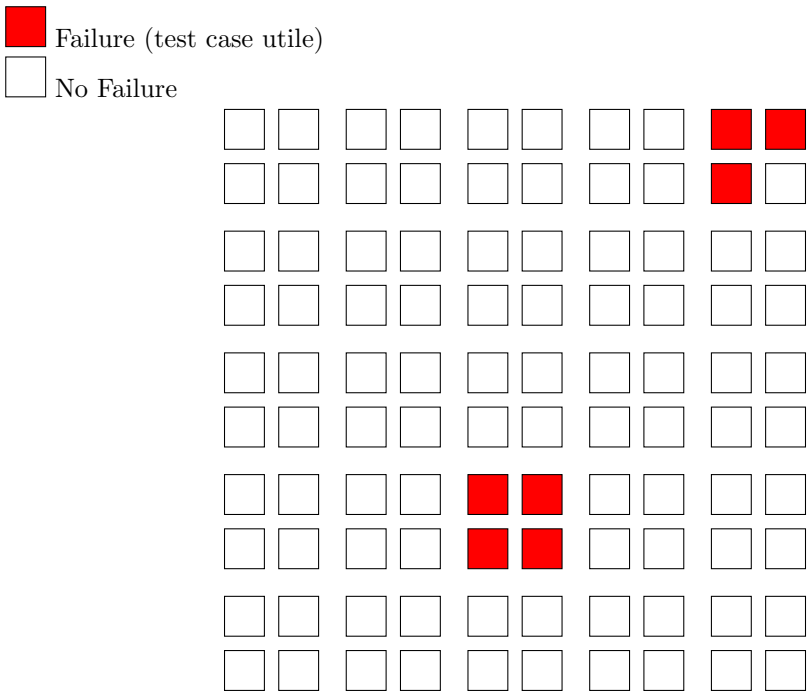
Esempio 3.1

Si supponga di dover testare un programma che risolve un'equazione di secondo grado, che sappiamo essere risolvibile con la seguente formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

L'input saranno i coefficienti a, b, c ed il programma ritorna $x_{1,2}$.
In questo caso, andare a testare esempi random non è l'approccio più corretto da fare, poiché non si vanno ad esaminare i casi più di interesse, come quando $\Delta = \sqrt{b^2 - 4ac} = 0$, oppure $2a = 0$.

Figure 5: Distribuzione delle failure



Nella figura sopra, vediamo come le failure siano scarse in tutto il range di input, ma si addensino attorno a certi punti, che sono i nostro punti di interesse.

In generale, si vogliono eseguire poche prove, ma fondamentali. Ovviamente, è impossibile stabilire a priori, dato che gli input sono infiniti, dove si verificheranno esattamente le failure. Possiamo, però, dividere gli input possibili in insiemi, e per ciascuno di questi testare almeno un input, in maniera tale da poter coprire la maggior parte dei casi; alternativamente, si può ragionare a partire dal codice, supponendo che ogni statement corrisponda ad una classe di input. Questi, ed altri criteri per la scelta dei test, vengono definiti *criteri di adeguatezza*.

3.1 Criteri di adeguatezza

Esistono diversi tipi di criteri di adeguatezza, ed ogni tipologia ha un'origine diversa.

Criteri funzionali:
derivano dalle specifiche del software

Criteri strutturali:
derivano dall'analisi del codice

Criteri basati sui modelli:
i modelli vengono estratti dalle specifiche (test su un protocollo di comunicazione)

Criteri basati sui fault:
si ipotizza che il programma possa contenere certe classi di fault (come per esempio il buffer overflow)

3.1.1 Funzionali vs Strutturali

In generale, nessuno dei due criteri prevale sull'altro, infatti spesso si adotta una combinazione dei due.

Esempio 3.2

Supponiamo di avere il seguente frammento di codice:

```
int fun (int param) {
    int result;
    result = param / 2;
    return result;
}
```

Ed analizziamolo sia dal punto di vista della specifica, che dal punto di vista del codice:
Specifica: un programma che prende in input un intero n e restituisce n / 2 se l'input è un numero pari, o solo n se l'input è un numero dispari.

Figure 6: Criteri a confronto

Criterio basato sulla specifica	Criterio basato sul codice
Sono necessari almeno due test: • Almeno un test per input pari • Almeno un test per input dispari	È sufficiente un unico test per eseguire tutti gli statement

Nel caso del *criterio basato sulla specifica*, si rivelerà un malfunzionamento durante il test con input dispari.
Con il *criterio basato sul codice*, non avrò la certezza di individuare il bug, poiché potrebbe essere testato un singolo numero pari.
Quindi è chiaro, in questo caso, che la scelta migliore sia di adottare un *criterio basato sul codice*; si può inoltre notare come in questo esempio venga evidenziata la mancanza di implementazione di una parte della specifica (errore di *missing logic*).
In generale, il *criterio basato sulla specifica* ci aiutare a trovare facilmente errori di *missing logic*.

3.2 Combinational Testing

4

5