Software testing

Introduzione

Obiettivi:

- Comprendere l'importanza del software testing all'interno delle fasi di sviluppo del sistema
- Adeguare propriamente metodiche di testing affinchè sia acquisibile un certo livello di confidenza con il sistema software elaborato

Parte cruciale all'interno di un qualsiasi processo di sviluppo software consiste nella creazione di *testing tools*, i quali provvedono a specificare il livello di *correttezza* e di *validità* del progetto implementato. L'etimologia del termine *testing* non deve essere associata a contesti inerenti alla qualità del codice, i propri obiettivi sono del tutto differenti, affini alla creazione di un concreto livello di *confidenzialità* capace di esprimere se la struttura implementata all'interno del sistema sia corretta o meno.

Rispetto a quanto detto si analizzano due tematiche inerenti alla breve descrizione precedente, in cui il termine *software testing* è accomunato da una duplice espressione, ossia:

Definizione informale
Il termine *software testing* promuove due caratterizzazioni suddivise in *validità* e *veridicità*; il primo vocabolo esprime la valutazione attuata per definire la bontà dell'architettura del sistema software, mentre il corrisposto propone una stima della correttezza dell'implementazione adeguata fino ad ora.

Concludendo l'intento di un meccanismo simile consiste nella rivelazione del maggior numero di *diffetti*, poichè il costo relativo alla correzione di situazioni errate è proporzionalmente diretto al tempo di risoluzione. Riassumendo le fasi di *test* sono fondamentali per veicolare il progetto verso un vantaggio competitivo, ma ciò può avvenire solamente se tali comportamenti siano immediatamente attuati e ripetitivi nel *lifecycle* della soluzione.

Sintassi

Di seguito è riportato l'insieme di definizioni che contraddistinguono il contesto descritto.

Definizione defect
Un *defect*, detto anche *bug*, indica un errore logico, che non sempre si traduce in un concreto malfunzionamento.

Per cui un difetto è una sequenza di istruzioni, che, quando eseguite con particolari dati in input, genera un *malfunzionamento*, ossia un comportamento software difforme dai requisiti espliciti. La mancata effettività del malfunzionamento avviene qualora non siano immessi i dati in input tali da evidenziare l'errore all'interno del codice che contiene il diffetto.

Definizione failure

Una *failure* rappresenta il momento in cui viene riscontrato concretamente il *defect*.

Tradotto, un *malfunzionamento* indica un comportamento contrario all'aspettative, in cui, come già detto prima, si riscontra qualora programmatori o sviluppatori immettino un errore all'interno della soluzione provocando un *bug*, il quale diviene una *failure* solamente se poste certe condizioni.

Definizione test case
Test case rappresenta l'elenco che contiene una breve descrizione degli *outcome* attesi da singoli processi esecutivi, i quali includono *parametri*, *condizioni* e *risultati* attesi. Nel gergo l'insieme dei *test case* da vita a *test set*.

Testing levels

I test possono essere adeguati a differenti granularità, suddividendosi in *unit testing*, *integration testing* e *end-to-end testing*. Oltre al grado di precisione che contraddistingue ognuno di essi, sono caratterizzati da una specifica complessità, in cui è possibile porre al primo posto la tipologia *end-to-end*. Di seguito è proposta una visione più dettagliata di ogni caratterizzazione:

- *Unit testing* riferisce a un meccanismo di verifica che opera su *specifiche sezioni* del *code base*, solitamente relative a livelli funzionali e operativi. Tendenzialmente queste tipologie di test sono adoperate da sviluppatori in concomitanza alla stesura di metodi, affinchè possano assicurarsi che operino correttamente. Tuttavia *unit testing* non è in grado di analizzare la correttezza delle sezioni software prese in considerazione, ma stabilisce se i blocchi di codice possono svolgere in maniera indipendente le proprie funzionalità; si ricorda l'importanza delle *dipendenze* all'interno di uno sviluppo software, capaci di inibire l'intera struttura ideata qualora sia erroneamente gestita.
- *Integration testing* tenta di controllare attivamente il livello di astrazione posto tra classi legate alla logica algoritmica ed entità inclini ad interfacce utente. Per cui, un meccanismo di controllo simile pone la propria attenzione sulla ricerca di *defects* posti tra interazioni di elementi che compongono il modello analizzato.
- *End-to-end testing* rappresenta lo strumento di analisi più complesso e costoso in assoluto, poichè tenta di controllare l'intero sistema software attuato. Solitamente non sono riscontrabili test automatizzati simili, data l'elevata difficoltà nella creazione di un tool in grado di analizzare ogni singolo dettaglio che caratterizza il progetto osservato, a causa di questa principale ragione non risulta essere molto adoperato all'interno di team di sviluppo.

Testing approach

Sono presenti differenti approcci che possono essere adeguati in fasi di *software testing*, suddivisi in due famiglie, *dynamic* e *static*. Le due metodologie illustrate sono strettamente correlate, in cui la prima citata è attuata affinchè sia possibile *eseguire* il *code base* pur di visualizzare i *bug*, mentre la corrisposta è adoperata per *analizzare* il *code base* pur di individuare i *defects*. Entrambe condividono la stessa finalità anche se spesso sono valorizzati approcci *statici* che *dinamici*; la ragione è dovuta alla netta semplicità che condividino le

prime metodologie rispetto alle seconde introdotte.

Data la netta supremazia degli approcci *statici*, di seguito sono proposti alcuni dei più diffusi, quali:

- *Model checking*, ...
- *Symbolic execution*, ...
- *Data-flow analysis*, ...
- *Abstract interpretation*, ...

Black-box testing

Il metodo *black-box* rappresenta uno strumento di analisi delle funzionalità di un sistema software, dove la propria particolarità deriva da una mancata conoscenza dell'implementazione interna dell'architettura osservata. Perciò pone la totale attenzione sulla bontà del risultato finale, piuttosto del procedimento attuato per risalire ai dati evidenziati; grazie a questo livello di caratterizzazione un metodo simile può essere adeguato all'interno di un qualsiasi *testing layer*, quindi ad una qualsiasi granularità.

Come detto in precedenza, non sono richieste conoscenze legate alle specifiche del codice, poichè il *tester* è formulato in modo che conosca *cosa* il sistema software dovrebbe fare, ma non su *come* sia il procedimento che porti al risultato tangibile.

Il termine *black-box testing* raffigura l'insieme di approcci simili, dove uno dei più diffusi consiste:

- *Boundary value analysis*, illustra una tecnica di *software testing* incline alla visualizzazione del solo risultato; come da denominazione, l'implementazione del meccanismo prevede l'utilizzo di limiti numerici, detti *punti di discontinuità*, su cui verte l'intera struttura operativa.

  Il test sarà applicato fornendo in input valori strettamente superiori e inferiori ai *boundary values*, da cui si osserveranno i comportamenti attesi rispetto ai dati introdotti. **Esempio slide 13**

White-box testing

Contrariamente a *black-box*, l'approccio *white-box* è più incline alla costruzione di test affini alla struttura interna del codice del sistema software analizzato. Per cui pone maggiore attenzione sulla bontà dell'architettura che abbia portato alla realizzazione di dati finali, piuttosto di valorizzare un meccanismo di verifica che si sofferma solamente sul risultato in sè.

Anche in questo contesto, *white-box* si suddivide in una serie di caratterizzazioni, in cui alcune di esse risultano:

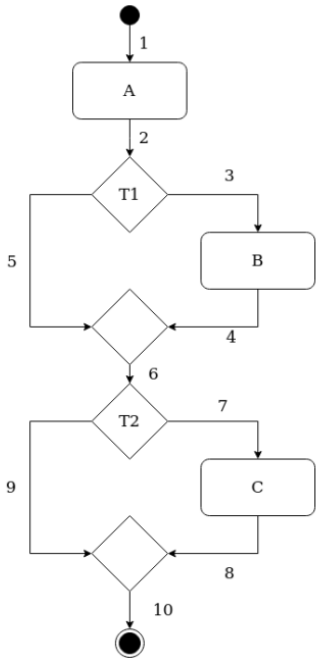
- *Code coverage*, prende in esame solamente il percorso che compone il corretto raggiungimento all'entità finale

- *Branch coverage*, raffigura l'analisi sia del percorso ritenuto corretto, ma anche del proprio opposto qualora non sia verificate certe condizioni all'interno di nodi decisionali
- *Path coverage*, considera ogni singolo percorso che possa concretizzarsi all'interno della struttura del sistema software

Nonostante le differenze poste, è bene soffermarsi sul termine che accomuna ognuno di tali approcci, ossia *coverage*, in cui l'obiettivo consiste nell'instaurazione di maggiore confidenza con il *sistema software* sviluppato ad ogni interazione, che si pone oltre alla semplice verifica di validità della soluzione proposta.

Di seguito è proposto un esempio affinchè sia possibile comprendere all'atto pratico modalità d'uso di strumenti simili.

Caso di studio



Lo schema proposto raffigura il processo esecutivo di un certo blocco di codice posto all'interno del sistema software, il quale è stato rimpiazzato mediante alcune notazioni grafiche per rendere al meglio la visualizzazione di un percorso. Dal *flow chart* presentato, si articolano un totale di sette percorsi distinti, raggruppati in tre insiemi differenti, inerenti alle descrizioni di *coverage* precedenti. Esempi di quanto detto sono:

- *Code coverage*
  - 1;A;2;3;B;4;6;7;C;8;10
- *Branch coverage*
  - 1;A;2;3;B;4;6;7;C;8;10
  - 1;A;2;5;6;9;10

- *Path coverage*
  - 1;A;2;3;B;4;6;7;C;8;10
  - 1;A;2;5;6;9;10
  - 1;A;2;5;6;7;C;8;10
  - 1;A;2;5;6;9;10

Path coverage è sicuramente il meccanismo più dispendioso dei tre, e data la complessità che lo caratterizza si riserva solamente per punti critici, anche se questo comporta, da canto suo, ad un elevato dispendio in termini di mantenimento e stesura del codice.

Black vs White

Black-box e *white-box* differiscono principalmente sull'approccio adoperato per garantire il giusto livello di confidenzialità, il primo incline al risultato tangibile mentre il secondo affine alla bontà della struttura interna del sistema software. In relazione a quale delle due risulti migliore non è possibile rispondere, poichè sono caratterizzate entrambe da imperfezioni e priviliegi. Perciò sono illustrate le peculiarità di ognuno dei meccanismi.

Black-box testing

- *Pro*
  Black-box testing si fonda solo sul *comportamento* della soluzione software attuata, in cui non è imposta la comprensione del *code base* ma di approcciare un giusto livello di logica algoritmica affinchè si possa riconoscere il *behavior* atteso. Spesso i test sono implementati in modo da riflettere i soli requisiti funzionali, da cui risulterà immediata la comprensione di quale funzionalità non eroghi correttamente l'*output* desiderato, solo se adoperata la corretta struttura di *isolamento* tra entità del sistema software.
- *Cons*
  Data la costruzione di un insieme di *test* legati al *comportamento* atteso non è garantito un ottimo livello di copertura, a causa del livello alto di analisi imposto dall'approccio *black-box*.

White-box testing

- *Pro*
  Approcci correlati al metodo *white-box* garantiscono l'instaurazione di un elevato livello di *confidenzialità*, poichè stabiliscono la costruzione di test sull'architettura del *code base*, da cui deriva la necessità di studiare e comprendere le particolarità della soluzione sviluppata.
- *Cons*
  Nonostante esprima un *coverage* estremamente ampio, comporta ad un numero crescente di percorsi possibili, i quali possono aumentare in maniera smisurata e, pertanto, provocare un grande dispendio di risorse. Il codice per un test simile è piuttosto difficile da implementare e da mantenere.

Test the tests

La bontà di un *test* è la correttezza in cui sia possibile analizzare e rilevare *difetti* che si possano tradurre in *malfunzionamenti* all'interno del *software system*. Tuttavia, la difficoltà principale consiste nella valutazione dei test sviluppati in maniera tale da osservare il grado di validità correlato.

L'effettività desiderata da un qualsiasi team di sviluppo prevede la realizzazione di test di pregievole qualità in grado di elencare tutti gli aspetti errati del sistema software; ma come è possbile stabilire se un codice di verifica sia ben strutturato?

Ciò avviene tramite la creazione di *mutanti*, ossia versioni del *code base* che contengono *errori* e *difetti*, affinchè si possano poi tradurre in *malfunzionamenti*. Ottenuto in *input* il *mutant*, il *software testing* ideato dovrebbe individuare i *bug* e renderli visualizzabili ad attori esterni, garantendo in questo modo codici di verifica corretti. Nel caso opposto questo potrebbe segnalare equivoci effettuati all'interno della soluzione.

Isolamento

Gli approcci descritti fino ad ora possono essere adoperati con tre modalità di precisione, in cui il più utilizzato appartiene a *unit testing*. Dato che opera su singola unità, come classi e metodi, si fonda sul concetto di *isolamento*.

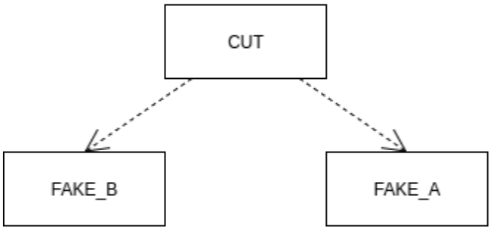
In relazione alla comprensione del concetto di *isolation* è bene soffermarsi su tre azioni sequenziali che potrebbero caratterizzare un comune processo di verifica:

- Avviene l'instanzazione dell'oggetto
- Si opera sull'oggetto affinchè sia manipolabile
- Si attuano funzionalità di verifica dello stato dell'oggetto dopo aver richiamato il metodo interessato

Nonostante spesso test del genere possano essere adeguati per analizzare la validità della soluzione software, non sempre sono in grado di indicare se *malfunzionamenti* siano imputabili sull'istanza in questione oppure a logiche interne; si ricorda la figura della dipendenza, in cui una classe software può richiamare funzionalità di altri elementi del modello per elaborare comportamenti interni. Rispetto a quando proposto occorre un **meccanismo di isolamento**.

L'implementazione di una tecnica simile avviene tramite l'impiego di *controfigure*, le quali forniscono automaticamente i parametri necessari a classi analizzate e devono essere sviluppate tramite una minima logica algoritmica, in modo tale che effettività errate non siano associabili alle nuove entità introdotte.

Caso di studio



Nell'esempio proposto si osservano le due entità *FakeA* e *FakeB*, le quali rappresentano le controfigure delle classi concrete che riportano i dati necessari per inizializzare le fasi di test. Tuttavia, essendo elementi del modello legati a logica algoritmica, l'assenza di un livello di astrazione potrebbe disorientare le fasi di verifica attuate.

La soluzione prevede l'inserimento di interfacce, poste tra classe verificata e dipendenze della singola unità modellativa, poichè in totale assenza non sarebbe possibile associare l'errore riscontrato. Concludendo, occorre che anche codici dei test siano relativi a un layer minimo di qualità, dato che contribuiscono ad un maggiore isolamento dei termini e su cui è possibile, conseguentemente, adeguare *pattern* di differente natura.