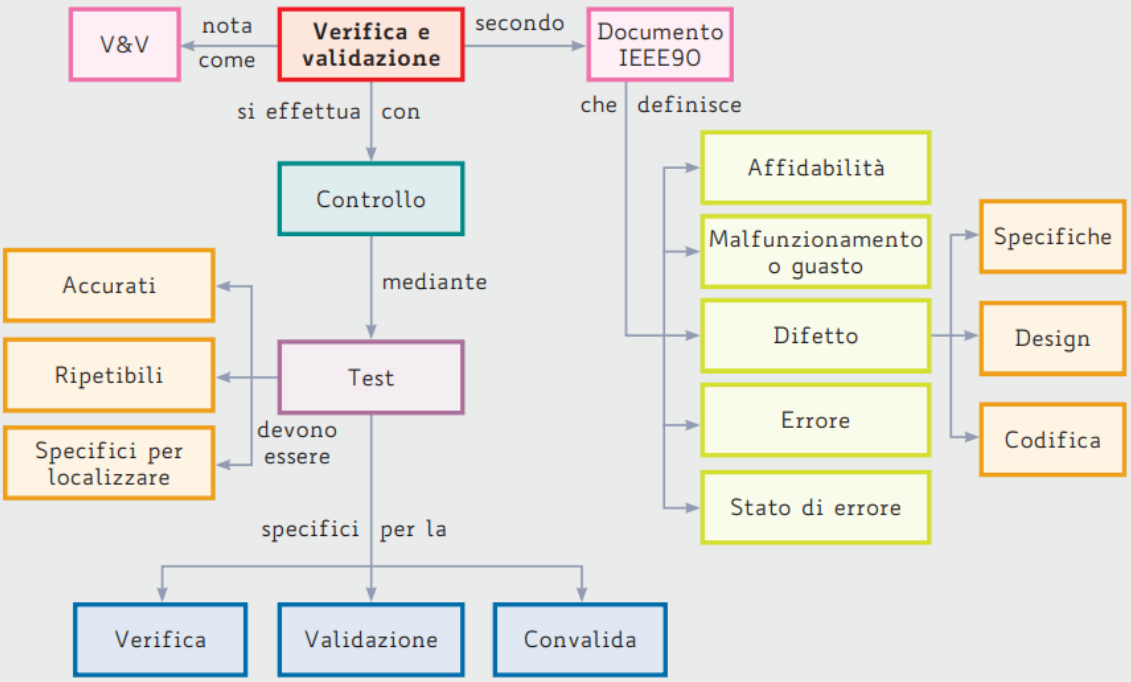
1 Verifica e validazione del software



Introduzione

In questa Unità affronteremo il delicato problema del controllo del software, costituito da tutte le attività inserite generalmente alla fine del processo di sviluppo, con il duplice scopo di verificare il corretto funzionamento del sistema e la rispondenza dello stesso alle aspettative del committente, garantendo un prefissato livello di qualità del prodotto.

Con il termine controllo si intende un insieme complesso di operazioni che sono la verifica, la validazione e la convalida mediante un insieme di operazioni di test.

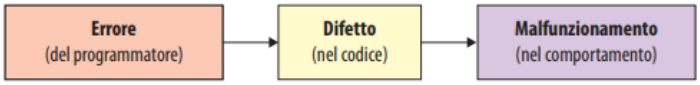
IEEE 90
Institute of Electrical and Electronics Engineers – *"IEEE Standard Computer Dictionary – Standard Glossary of Software Engineering Terminology"*, approvato il 28 settembre 1990.

Terminologia di verifica del software

Prima di affrontare questo argomento riportiamo il glossario dei termini adottato, che fa riferimento al Documento **IEEE 90** che propone le seguenti definizioni (fra parentesi diamo il termine originale inglese):

- affidabilità (reliability): la misura di successo con cui il comportamento osservato di un sistema è conforme a una certa specifica del relativo comportamento;
- malfunzionamento o guasto (failure): qualsiasi deviazione del comportamento osservato dal comportamento specificato;
- difetto (bug/fault): la causa meccanica o algoritmica di un errore;
- errore (error): è l'origine di un difetto;
- stato di errore (error status): il sistema è in uno stato tale che ogni ulteriore elaborazione da parte del sistema porta a un fallimento.

Possiamo quindi rappresentare schematicamente la relazione tra errore e malfunzionamento con il seguente disegno:



Requisiti di qualità dei test

Il testing consiste nell'esecuzione delle operazioni atte a trovare le differenze tra il comportamento atteso e quello effettivo; possiamo evidenziare tre caratteristiche che deve avere:

1 i test dovrebbero essere accurati e il loro successo deve essere valutato in un modo obiettivo: sono da evitare situazioni in cui il risultato di un test potrebbe essere un'immagine che deve essere interpretata, quindi, suscettibile a giudizio parziale di un essere umano;

2 i casi di test dovrebbero essere ripetibili: come per la fisica, dove l'esperimento deve essere riproducibile in condizioni controllate, è necessario che anche per l'informatica si sia in grado di poter ripetere lo stesso test nelle identiche condizioni, sia software che hardware; ma *"è sempre possibile garantire le stesse condizioni in caso di situazioni critiche dovute alla concorrenza e alla competizione tra i processi?"*;

3 i test dovrebbero aiutare nella localizzazione dei guasti: è necessario orientarsi nella realizzazione di tanti casi di test semplici che sono migliori di pochi test articolati, in quanto un caso di test complesso ha una probabilità maggiore di fallire a causa della presenza di più di un difetto e potrebbe non aiutare all'individuazione di nessuno di essi.

Verification & Validation

Abbiamo detto che la fase di V&V viene effettuata dopo aver "terminato la realizzazione del prodotto" con il fine di capire se il sistema risolve effettivamente i problemi per cui è stato concepito e se lo fa correttamente.
Barry W. Boehm nel suo articolo del 1979 poneva le domande che seguono.

Verification: *"Have we built the product right?"*

Validation: *"Have we built the right product?"*

che possiamo tradurre con:

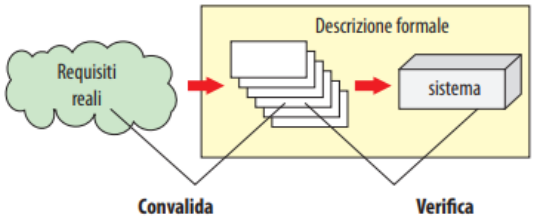
Verifica: *"Il prodotto che stiamo sviluppando è corretto?"*

Validazione: *"Stiamo sviluppando il corretto prodotto?"*

Possiamo quindi sintetizzare questa fase con la ricerca delle risposte alle seguenti domande:
A per la verifica:
- *Il software rispetta le specifiche?*
- *È stato implementato tutto quanto descritto nel documento dei requisiti?*
- *Ho implementato correttamente tutte le funzionalità?*

B per la validazione:
- *Il software rispetta ciò che voleva il cliente?*
- *I requisiti modellano ciò che il cliente realmente voleva?*

Nella validazione, anche detta convalida, si eseguono i test di usabilità, dove l'elemento fondamentale è il feedback dell'utente: viene infatti convalidato il soddisfacimento dei requisiti che erano alla base delle aspettative del cliente.

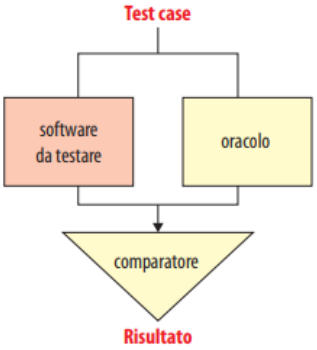


Test e Casi di test

Il processo di verifica e validazione dovrebbe essere applicato a ogni fase durante lo sviluppo e non solo al termine dello sviluppo: quando si effettua il test del componente (component testing) l'obiettivo è quello di collaudare le singole parti/unità elementari del programma mentre nei test di sistema (system testing) si deve anche collaudare il funzionamento integrato di gruppi di componenti per il fine funzionale di un sistema o sottosistema.

L'oracolo

Per potere effettuare un test è necessario essere in grado di valutare la correttezza o meno del suo risultato: quindi la condizione necessaria per effettuare un test è quella di conoscere in "anticipo" il comportamento atteso in modo da poterlo confrontare con quello osservato.



> Viene indicato col termine oracolo il risultato atteso per ogni singolo test case.

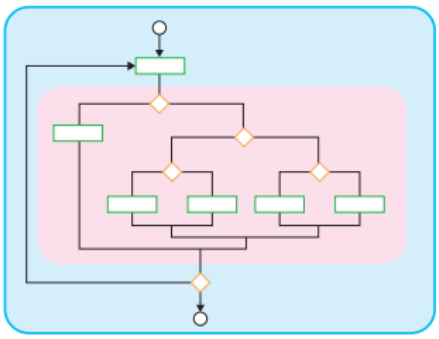
Spesso l'occhio umano non è lo strumento migliore per osservare il risultato, cioè per effettuare la comparazione, anche perché a volte possono essere necessarie migliaia di dati in input per ciascun test presente nel test case: ci si appoggia su strumenti automatici, cioè su oracoli automatici, generati da specifiche formali, che eseguono batterie di prove, composte a partire da quelle effettuate nel collaudo della versione precedente del sistema.

Ma **quante** devono essere le valutazioni affinché un test sia esaustivo?

Cerchiamo di rispondere a questo interrogativo analizzando un semplice esempio di segmento di programma.

ESEMPIO

Supponiamo che il ciclo esterno di questo segmento di codice venga eseguito 200 volte.



È facile verificare che sono possibili circa 10^{14} alternative di flusso di esecuzione e, se ipotizziamo di eseguire un'operazione di test in un microsecondo per avere la certezza che in ogni condizione il programma sia corretto, un test automatico ci impiegherebbe circa 3170 anni.

Non è quindi possibile effettuare sempre tutti i test: l'importante è scegliere al meglio i casi da valutare.

Il debugging

In caso di esito negativo di un test, cioè se ci si trova nella situazione in cui il comparatore rileva una difformità dal valore atteso, sicuramente il segmento/l'unità di codice analizzati nasconde un difetto.

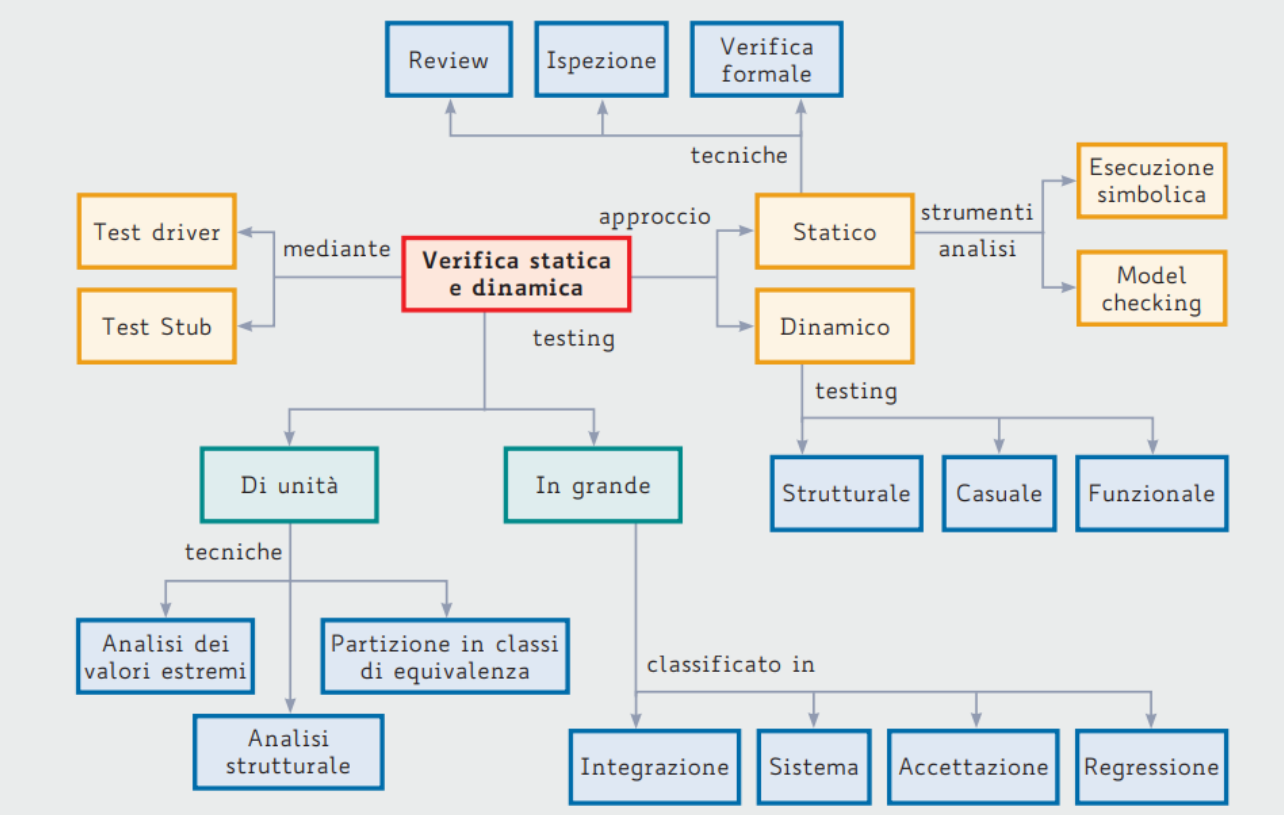
Naturalmente il difetto deve essere rimosso: la procedura che viene attivata a tal fine prende il nome di debugging.

Letteralmente il termine debugging significa togliere da un programma i bug ("baco") che ne impediscono la corretta esecuzione, dove con bug si intendono gli errori software.
Il termine bug ha un'origine "storica", in quanto nei primi calcolatori spesso la causa di malfunzionamento era dovuta proprio a insetti che facevano il nido tra le memorie magnetiche, oppure al caldo delle valvole, tecnologie che oggi sono state completamente soppiantate dai circuiti integrati.
Il termine è stato coniato da Grace Murray Hopper, che scoprì nel 1947 un errore del calcolatore MARK II, uno dei primi elaboratori nella storia dell'informatica, dovuto proprio a un insetto che aveva mandato in crisi un circuito (una falena si era incastrata in un relè).
Anche per il debugging sono state raccolte e descritte tecniche specifiche; inoltre, integrati negli ambienti di sviluppo, sono disponibili strumenti appositi per effettuare questa operazione: i debugger.
Il debugger consente di interrompere l'esecuzione del programma in corrispondenza di particolari istruzioni "marcate" come **breakpoint**, quindi di controllare il contenuto delle variabili in memoria in quel preciso passo; infine permette anche di modificare "manualmente" il contenuto delle variabili.

Una funzionalità aggiuntiva offerta dai debugger è l'esecuzione controllata del codice che può essere effettuata a partire da un breakpoint: il programmatore con questa opportunità è in grado di seguire passo passo l'esecuzione del programma e monitorare il valore delle variabili, individuando anche il punto esatto dove il programma "genera" l'errore.
Una volta localizzato il bug si procede alla sua eliminazione e, naturalmente, alla ripetizione dei test che ne hanno permesso l'individuazione per verificare ora la sua assenza: le difficoltà nella verifica le troviamo in presenza di sistemi concorrenti o real time.

Breakpoint
Breakpoint (punti di interruzione): corrispondono a righe di codice sorgente scelte dal programmatore su cui l'esecuzione dell'applicazione viene sospesa dal debugger.

2 Verifica statica e dinamica



Verifica statica

Gli obiettivi di base nella verifica e validazione sono di identificare e risolvere i problemi che hanno un alto rischio di compromettere sin dall'inizio il ciclo di vita del software.
Come detto in precedenza, dobbiamo individuare due momenti e due modalità differenti per effettuare la V&V e le tecniche di verifica possono essere divise in due macro-categorie:

- approccio statico, anche detto ispezione del software;
- approccio dinamico, effettuato avviando il sistema attraverso dati di test.

La verifica statica è un processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto, documentazione, che viene effettuata senza ricorso alla esecuzione del codice.
Tra le tecniche utilizzate nella verifica statica ricordiamo il review, l'ispezione, la verifica formale, l'esecuzione simbolica ecc. che possono essere suddivise in due categorie:

A verifiche di coerenza, come le verifiche di sintassi, la tipizzazione e la corrispondenza dei parametri fra procedure diverse;

B tecniche di misurazione, che includono verifiche di leggibilità del codice e livelli di struttura.

Le review o desk check

Il metodo classico per effettuare il controllo statico è quello storico, utilizzato dai programmatori degli anni Ottanta, che si basa sull'analisi informale del codice, effettuando "a mano" la simulazione della sua esecuzione: a questo metodo di review viene dato il nome desk check.

Anche se il controllo statico è sinonimo di attività manuale, nella pratica molti controlli statici possono essere eseguiti con l'ausilio di analizzatori di codice creati specificamente per i diversi linguaggi.

Tool di analisi statica

Ce ne sono tantissimi, ognuno con specifici scopi.

Esempi in java:

* Checkstyle che concentra le sue analisi su regole stilistiche e di formattazione
  - Findbugs è uno strumento sviluppato dall'Univeristy of Maryland
    - Si concentra soprattutto sulla ricerca di pratiche di programmazione scorrette che potrebbero essere causa di bug
      - *Ovviamente essendo uno strumento di analisi statica non può in generale dimostrare che si tratti di bug non potendo osservare alcun fallimento*

Model checking (controllo sul modello)

Il model checking può essere utilizzato nei casi in cui il linguaggio per descrivere il comportamento del sistema (il "modello") e le sue proprietà che devono essere verificate è sufficientemente espressivo.

Programmi che generano codice a runtime

In alcuni casi è impossibile effettuare la verifica statica del codice in quanto esso viene generato runtime dal programma stesso: ricordiamo per esempio come una pagina server può generare codice lato client, come per esempio le pagine php generate come risposta a un'interrogazione.
Anche una pagina Javascript può essere generata dinamicamente e può essere diversa a seconda dei dati e delle situazioni: in questi casi è impossibile effettuare il test statico e, quindi, l'unica possibile verifica di queste situazioni è quella dinamica.

Verifica dinamica

A differenza del controllo statico, il controllo dinamico richiede l'esecuzione del programma da verificare fornendogli opportune "batterie" di dati di ingresso: è quello che corrisponde a ciò che comunemente viene chiamato test.
Una possibile definizione di testing dinamico è riportata di seguito.

Testing dinamico
Verifica dinamica del comportamento del software rispetto a quello atteso, utilizzando un insieme finito di casi di test, appropriatamente selezionati nel dominio di tutti i casi possibili nel dominio applicativo:
– dinamico: perché il test prevede l'esecuzione del programma;
– atteso: perché dalle specifiche deve essere possibile conoscere a priori il comportamento del software nei casi di test;
– finito: perché, in pratica, è possibile sottoporre il software a pochi casi di test;
– selezionati: perché le tecniche consentono di scegliere i casi che partecipano alla verifica.

I metodi disponibili per eseguire la verifica dinamica possono essere classificati in tre categorie:

1 testing funzionale: vengono collaudate tutte le funzionalità previste dalle specifiche identificate dai requisiti e realizzate nel sistema; il classico esempio di test funzionale è noto come *collaudo* a scatola nera o black test: si considera l'elemento da testare come una black box e viene inserito come input un insieme di valori che attivino tutte le transazioni previste per quella porzione/sezione di sistema;
2 testing strutturale: viene collaudato il funzionamento interno del software; questo prevede una conoscenza totale della sua architettura e delle scelte implementative e di design. Il tipico esempio di test strutturale è il collaudo a scatola bianca o white test: si considera l'elemento da testare come una white box della quale si vogliono verificare tutti i dettagli implementativi sempre tramite l'input di dati che ne verificano, possibilmente, tutti i percorsi e le alternative possibili.
3 testing casuale: dopo aver predisposto un'opportuna collezione di test, ne viene selezionato un sottoinsieme tramite una generazione algoritmica pseudocasuale che viene effettuato sul sistema.

Testare tutte le possibili situazioni, cioè effettuare quello che prende il nome di collaudo esaustivo, che necessiterebbe di provare tutte le possibili combinazioni dei dati di input, rientra nella categoria del testing casuale, ma, nella maggior parte dei casi, è impossibile da raggiungere.

In questa Lezione introduciamo gli aspetti principali di questa tecnica di verifica a partire dalla sua pianificazione in quanto è di fondamentale importanza studiare e individuare cosa va testato e quale input inserire per provocare un malfunzionamento; per essere efficace un test deve essere realizzato in base alle caratteristiche del programma da esaminare e fatto in modo da valutarne le criticità.
Dobbiamo comunque sempre tener presente la tesi di Dijkstra che, benché abbastanza datata (1972), anche oggi ha la sua assoluta validità.

Tesi di Dijkstra
Il testing di un programma può rilevare la presenza di malfunzionamenti ma mai dimostrarne l'assenza.

Il testing funzionale e quello strutturale sono tra loro complementari: riportiamo in una tabella le loro caratteristiche in modo da poterle confrontare.

BLACK BOX	WHITE BOX
Non necessita della conoscenza del codice sorgente.	Richiede la conoscenza del codice sorgente.
È fortemente influenzato dalla notazione delle specifiche.	È basato sulla copertura del flusso di dati o delle istruzioni di controllo.
È scalabile.	Non è scalabile.
Non distingue fra implementazioni diverse corrispondenti alla stessa specifica.	Non riesce a rivelare omissioni in caso di alternative delle specifiche che non sono state implementate.
I test case sono derivati dalle specifiche di una unità.	I test case sono derivati dalla struttura interna di una unità.

Livelli del testing

Abbiamo detto che il testing consiste nel trovare le differenze tra il comportamento atteso e quello effettivo e, per avere un miglior risultato, è opportuno che venga effettuato da persone che non sono state coinvolte nelle attività di sviluppo del sistema.
Le operazioni di testing vengono effettuate a vari livelli, anche in base alle dimensioni del sistema: in ogni caso qualunque prodotto è composto da un insieme di componenti integrati e cooperanti e, quindi, le tecniche di testing si concentrano sull'analisi dei singoli componenti per poi arrivare al test dell'intero sistema seguendo strategie di integrazione dettate dall'architettura.
Possiamo distinguere due livelli di testing:

1 test di unità (unit test): di competenza dello sviluppatore, verifica il funzionamento dei singoli moduli del sistema;
2 test in grande: di competenza di un'apposita squadra, verifica l'insieme di più moduli, dell'intero sistema e della eventuale "convivenza" con altri sistemi; può essere classificato in:
   a integrazione,
   b sistema,
   c accettazione,
   d regressione.

Unit testing

Definiamo come unit un insieme di uno o più componenti del software con associati dati di controllo, procedure di uso e operative.

Il test deve essere effettuato per ciascuna di queste unit che appartengano al sistema allo scopo di individuare almeno una tra le componenti nuove o modificate che non completi il test di unità negativamente.
Il test può essere effettuato utilizzando una oppure la combinazione di queste tecniche.

- Partizione in classi di equivalenza: viene effettuata dopo aver ripartito il dominio dei dati in input in sottoinsiemi tali che per tutti i valori di ogni sottoinsieme la unit si comporti allo stesso modo. Le classi di equivalenza possono essere definite come intervallo di valori, insieme di valori oppure valori legati a situazioni vincolanti o condizionate.
- Analisi dei valori estremi: si basa sulle classi di equivalenza ed esplora le situazioni sugli estremi e intorno agli estremi delle partizioni stesse.

- Analisi strutturale: è un white box test basato sui possibili percorsi effettuati dai dati all'interno della struttura della procedura; sono disponibili strumenti automatici di supporto alla generazione dei casi di test in grado di effettuare test specifici; come metrica di valutazione per indicare l'effettività dei test eseguiti viene utilizzato il concetto di copertura (coverage):
  – la sequenza delle istruzioni (statement coverage);
  – le condizioni (condition coverage);
  – i percorsi (path o branch coverage);
  – il flusso dei dati ("data-flow" coverage).

Modalità di esecuzione dei test

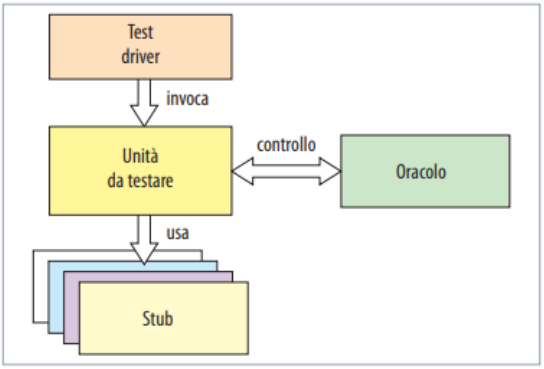
Il controllo su un modulo ha come obiettivo principale la correttezza funzionale delle operazioni esportate dal modulo a fronte della specifica definita nel progetto di dettaglio.

Inoltre per eseguire un test case su una unità (o una combinazione di unità) è necessario che questa sia isolata dal resto del sistema e, spesso, è necessario realizzare o simulare componenti che simulino le parti mancanti del sistema necessarie per avere una significatività del test.

Chiamiamo test driver e test stub i componenti creati ad hoc e usati per sostituire le parti mancanti del sistema: il loro insieme forma lo scaffolding, cioè l'ambiente per eseguire test case su una unità isolata.

Un test driver (o modulo guida) è un blocco di codice che inizializza e chiama la componente da testare: deve simulare l'ambiente chiamante e si deve occupare dell'inizializzazione dell'ambiente non locale del modulo in esame.

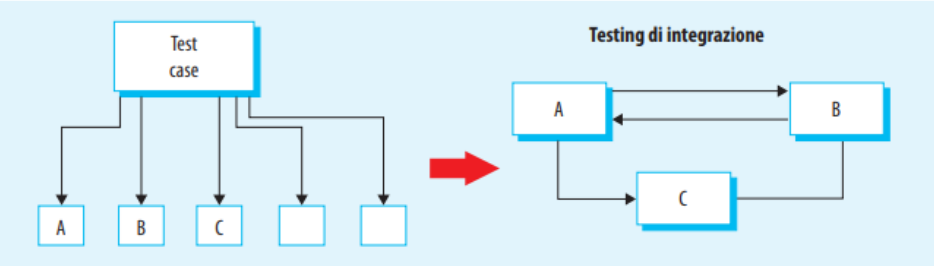
Un test stub (o modulo fittizio) è un'implementazione parziale di componenti da cui la componente testata dipende (componenti che sono chiamate dalla componente da testare); deve fornire la stessa API del metodo della componente simulata e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specificato nella signature.



La generazione di scaffolding può essere parzialmente automatizzata a partire dalle specifiche anche grazie a pacchetti software o plug-ins e add-on, disponibili per le più diffuse piattaforme di sviluppo come JUnit, NUnit, PUnit, CppUnit ecc.

Test di integrazione

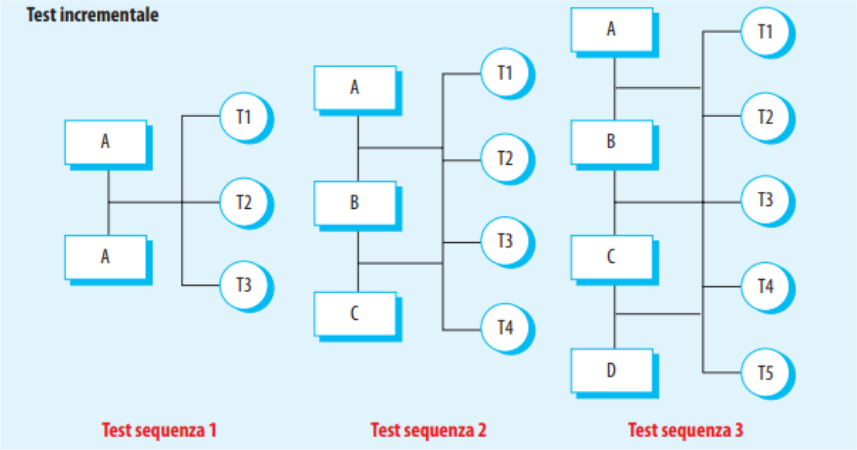
I test di integrazione (integration test): che sono necessari per verificare che le parti del sistema siano state collegate correttamente, la correttezza del programma nel suo complesso e l'assenza di anomalie sulle interfacce tra i moduli, devono essere effettuati sull'intera architettura del sistema; questi test vengono eseguiti solamente al termine dei test sui singoli moduli per validarne la loro funzionalità congiunta.
Per la loro attuazione è necessario progettare i casi di test necessari per provare il comportamento **di tutte le interfacce** tra componenti previste dal sistema, e per ciascuna di esse si devono definire i casi di test con le tecniche black box precedentemente descritte.



Obiettivi delle strategie d'integrazione sono la minimizzazione del lavoro e delle risorse necessarie all'integrazione e la massimizzazione del numero di difetti scoperti prima dei controlli sul sistema completo: alternare passi di integrazione a passi di controllo significa dover effettuare i test su sistemi che sono incompleti e, quindi, è necessario anche in questo caso che l'ambiente di test preveda stub e driver.
Ci sono due metodi per eseguire i test di integrazione, che si differenziamo per l'approccio.

1 Approccio non incrementale (big bang test): per effettuare questo test si deve aver completato i test isolati di unità per ogni modulo, prevedere di assemblare tutti i moduli e realizzare immediatamente l'analisi globale del sistema "senza una strategia specifica di integrazione", in analogia al big bang che ha creato l'universo; è comunque applicabile solo in sistemi di ridotte dimensioni.
2 Approccio incrementale: l'integrazione viene fatta in modo graduale, collegando uno a uno i singoli moduli e sostituendo gradatamente stub e driver con i moduli effettivi fino a costituire il sistema complessivo. Proprio per la sua natura incrementale è possibile effettuare l'integrazione tra i moduli man mano che questi sono

pronti, senza aspettare la terminazione di tutte le sottoparti, e ciò permette di individuare più agevolmente le anomalie presenti.



Possiamo classificare le modalità di approccio in base all'approccio con il quale vengono integrati i singoli moduli del sistema.

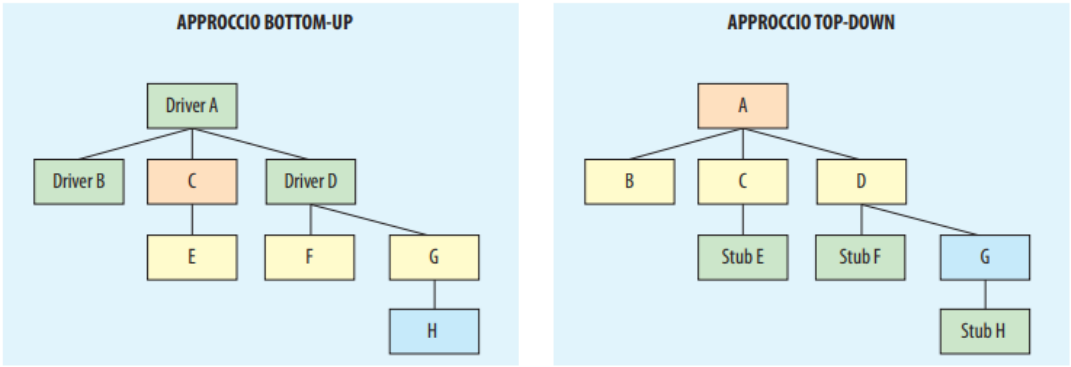
- Bottom-up: in questo approccio i moduli che non dipendono da altri moduli sono controllati singolarmente, partendo dal basso: si procede integrando i moduli collaudati risalendo l'albero, livello per livello, finché si raggiunge il modulo radice.

> ⚠ Sottolineiamo che per questa strategia l'esecuzione dei test ai vari livelli richiede solamente i driver.

- Top-down: si parte controllando il modulo radice e si procede integrando alla radice i moduli figli scendendo livello per livello fino a giungere alle foglie, effettuando in tal modo l'integrazione completa del sistema. Osserviamo che per ogni livello vengono ripetute le medesime serie di controlli, dato che l'interfaccia del modulo radice è sempre la stessa.

> ⚠ Dato che si parte dall'alto, per eseguire l'integrazione dei moduli sottostanti, sono necessari solo degli stub che però, soprattutto ai primi livelli, potrebbero essere in numero elevato e altamente complessi, in quanto devono simulare tutte le funzionalità del sistema.

- Sandwich: questa strategia è la più utilizzata nella pratica dello sviluppo software in quanto adotta contemporaneamente entrambe le strategie bottom-up e top-down con l'obiettivo di minimizzare i costi per la realizzazione di stub e driver.



Le più recenti metodologie di sviluppo prevedono di condurre in modo progressivo l'integrazione di un sistema, intervallando i vari passi con sessioni di test.

Test di sistema

Il test di sistema (system test) è la più classica attività di validazione che valuta ogni caratteristica di qualità del prodotto software nella sua completezza per effettuare un completo riscontro dei requisiti richiesti dell'utente; viene spesso effettuato da un team interno allo sviluppo al fine di approvare il rilascio del prodotto per la valutazione da parte del committente e avere la sua accettazione.
Le tecniche più adottate per i test di sistema sono basate su criteri funzionali con lo scopo di verificare il funzionamento del sistema nella sua interezza rispetto ai vincoli e ai requisiti espressi dal cliente.
Oltre ai fabbisogni del cliente, il test di sistema è necessario per verificare i requisiti non funzionali e che, quindi, sono anch'essi oggetto del test:

- test di stress (overload): viene condotto un test di funzionamento in condizioni di carico limite per numero di utenti e/o dispositivi connessi contemporaneamente: il sistema, anche al superamento del limite massimo di un percentuale aggiuntiva, non dovrebbe mai collassare in modo catastrofico;
- test di sicurezza: viene condotto un "attacco" alla sicurezza del sistema, cercando di accedere a dati o a funzionalità che dovrebbero essere riservate: lo scopo è di valutare il suo comportamento sia per casi dolosi che accidentali;
- test di robustezza: vengono inseriti nel sistema dati errati per osservare il suo comportamento;
- test di configurazione: nel caso in cui il sistema possa essere configurato in modi diversi e/o possa avere configurazioni hardware diverse, si verifica il suo comportamento in ogni possibile settaggio;
- test di usabilità: un aspetto fondamentale del sistema è che sia user-friendly, cioè abbia facilità d'uso per ogni utente, la sua documentazione sia semplice ed efficace e tenga conto dei livelli di esperienza e competenza dell'utente finale;
- test temporali (o di performance): sono necessari soprattutto per i sistemi che operano in tempo reale; si valuta l'efficienza dei tempi di elaborazione e dei tempi di risposta; la valutazione viene anche fatta con diversi livelli di carico e possono essere abbinati allo stress test;
- test di compatibilità: viene condotto un test per valutare la compatibilità del sistema con altri prodotti software, soprattutto per garantire la portatilità con le versioni precedenti soprattutto se deve sostituire o integrare un sistema esistente.

Test di accettazione (o collaudo)

Il test di accettazione (acceptance test) viene eseguito dal committente o da un suo delegato e deve verificare che il prodotto consegnato dal fornitore rispetti i requisiti richiesti: è spesso previsto nei contratti sotto la dizione collaudo ed è sempre presente nel caso di prodotto su ordinazione.

Specifica di test

* Documento che deve contenere almeno:
* **Il/I test bed (o bench) usati per ogni test**, cioè la piattaforma HW e SW di test, l'ambiente in cui il test deve essere effettuato, la preparazione dell'ambiente (installazione dei SW per esempio)
* **La procedura di test**:
  * Requisiti che vuole testare
  * Input iniziali
  * passi da effettuare descritti per esempio in una tabella come sotto
*

Passo	Azione	Risultato Atteso
1.		

Report di test

Documento che deve contenere almeno:

* Il test bed effettivo (per esempio il PC usato), data ed esecutore del test
* Gli input effettivamente dati ed i risultati che possono essere registrati aggiungendo 2 colonne alla tabella della specifica

Passo	Azione	Risultato Atteso	Input dati	Risultato
1.				

Cos'è il test di regressione

I test di regressione puntano a risolvere **un problema comune che gli sviluppatori devono affrontare** su base quotidiana: la comparsa di *regression bug* dopo l'**implementazione di una nuova funzionalità** o la **risoluzione di bug.**

Cos'è un regression bug?

È come se **il software regredisse, ritornando a uno stato in cui non funziona più correttamente**. È questo il caso in cui una funzionalità che si comportava correttamente in una versione software n, smette di funzionare nella versione n+1.

Per prevenire i regression bug, gli sviluppatori eseguono test di regressione.

L'obiettivo del regression testing è verificare che funzionalità sviluppate e testate in precedenza **funzionino correttamente, anche dopo aver apportato modifiche al codice**. I tester, di conseguenza, **eseguono i test esistenti sul codice modificato** per verificare se le nuove modifiche hanno distrutto qualcosa che funzionava prima dell'aggiornamento.

Il regression testing non è quindi un test una tantum, ma un processo ciclico che accompagna il software nelle sue diverse versioni.